Technische Universität Ilmenau

Fakultät für Elektrotechnik und Informationstechnik

Diplomarbeit

Entwicklung einer Clientsoftware zur Unterstützung des kontextsensitiven Routings

Inventarisierungsnummer: 2008-03-03/023/II00/2115

---

vorgelegt von:	Karsten Renhak
eingereicht am:	03. 03. 2008
geboren am:	14. 01. 1980 in Magdeburg
Studiengang:	Ingenieurinformatik
Studienrichtung:	Multimediale Informations- und Kommunikationssysteme
Anfertigung im Fachgebiet:	Kommunikationsnetze Fakultät für Elektrotechnik und Informationstechnik
Verantwortlicher Professor:	Prof. Dr. rer. nat. habil. Jochen Seitz
Wissenschaftlicher Betreuer:	Dipl.-Ing. Maik Debes

Kurzfassung

Ein individuell angepasster Austausch von Informationen gewinnt in heutigen Kommunikationsinfrastrukturen immer mehr an Bedeutung. Hierzu wurde vom Fachgebiet Kommunikationsnetze der TU-Ilmenau ein Ansatz des kontextsensitiven Routings entwickelt. Dieser lässt eine Beeinflussung der Wegewahl in paketvermittelten Netzwerken anhand von situationsgebundenen Eigenschaften und Umgebungsparametern zu. So können Nutzern auf ihre persönlichen Interessen zugeschnittene Informationen beziehungsweise Dienste angeboten werden.

Diese Arbeit beschäftigt sich mit der Erweiterung einer Testumgebung, welche der Verifikation kontextsensitiver Routingprotokolle dienen soll. Hierzu wurde eine Clientsoftware auf der Basis des „Ad-hoc On-demand Distance Vector Routingprotokolls“ (AODV) erstellt, die es Nutzern beziehungsweise Anwendungen erlaubt, kontextsensitive Dienste in Anspruch zu nehmen. Daneben stellt ein entworfenes Konzept zur transparenten Dienstnutzung durch gewöhnliche Anwendungen einen alternativen Implementierungsansatz vor.

Weiterhin befasst sich die vorliegende Diplomarbeit mit den Voraussetzungen einer Implementierung von reaktiven Routingprotokollen in gewöhnlichen Betriebssystemen. Hierzu wurden real existierende Implementierungen des reaktiven Routingprotokolls AODV untersucht und miteinander verglichen. Eine mögliche Erweiterung um kontextsensitive Routingfähigkeiten der bestehenden Implementierungen war Zielstellung des durchgeführten Vergleiches.

Sowohl theoretischen Aspekte als auch simulative Untersuchungen des kontextsensitiven Routings konnten mit Hilfe der implementierten Software im Rahmen der existierenden Demonstratorumgebung verifiziert werden. Die erzielten Ergebnisse liefern einen weiteren Beitrag für die praktische Umsetzung des kontextsensitiven Routings.

Abstract

Individual customised exchange of information in communication systems is gaining more and more relevance. Therefore an approach for a context sensitive routing mechanisms was developed at the Division of Communication Networks of the German University TU-Ilmenau. This approach affects a routing decision in a packet-switched network by situation based attributes and environmental parameters. The system offers each user an individual service which perfectly fits his needs and interests.

This thesis is extending an existing test bed for examining context sensitive routing protocols. On base of the "Ad hoc On-Demand Distance Vector Routing Protocol" (AODV) a client software was developed to allow users or applications to use context sensitive services. Moreover a draft about the transparent use of context sensitive services for common applications was presented.

Further conditions for implementing reactive routing protocols in common operating systems were presented. Implementations of the reactive routing protocol AODV were compared by the aim of extending context sensitive routing features.

The implemented software was able to verify theoretical aspects as well as simulation results from prior works about context sensitive routing. The gained results contribute real-life deployments of context sensitive routing networks.

Inhaltsverzeichnis

1 Überblick

Dieses Kapitel gibt dem Leser eine kurze Einführung in das Thema „Entwicklung einer Clientsoftware zur Unterstützung des kontextsensitiven Routings“ der vorliegenden Arbeit. Anschließend wird die konkrete Aufgabenstellung detailliert erläutert.

1.1 Einleitung

Unser Umfeld ist geprägt von Kommunikation und Dienstleistung. Kommunikation, auf Basis von technischen Hilfsmitteln durchdringt mehr und mehr den Alltag der Menschen. So können zum Beispiel die Nutzung des Internets oder der Besitz eines Mobiltelefons als Selbstverständlichkeit angesehen werden. Im Wandel der technischen Möglichkeiten entstehen neuartige oder veränderte Nutzungsgewohnheiten der Menschen im Umgang mit diesen Kommunikationsmöglichkeiten. So durchdringt die mobile Nutzung von Datendiensten zunehmend die Gesellschaft, wobei der Wunsch nach mehr Komfort für viele Anwender an erster Stelle steht.

Ein interessanter Ansatz neuartiger Kommunikationsverfahren geht davon aus, dass das Netzwerk, welches die Kommunikation ermöglicht, Eigenschaften des Nutzers selbständig auswertet und so personalisierte Dienste bereit stellen kann. Zur Personalisierung der angebotenen Dienste können sämtliche Informationen herangezogen werden, die etwas über eine Situation oder den Nutzer aussagen. Diese Informationen definieren den *Kontext* eines Nutzers. Man spricht auch von *kontextsensitiver* Dienstnutzung.

Das Fachgebiet Kommunikationsnetze der Technischen Universität Ilmenau entwickelte einen Ansatz zum kontextsensitiven Routings. Hierbei werden Entscheidungen über die Wegewahl in Netzwerken anhand von Kontextinformationen des Nutzers beeinflusst. Somit existiert ein Konzept, dass Parameter aus der Umwelt sowie Eigenschaften des Nutzers die Wahl gewünschter Dienste beeinflussen lässt.

Ziel dieser Diplomarbeit ist es, die bereits realisierten Teillösungen des Konzeptes zu ergänzen. Hierzu sind die bestehenden theoretischen Konzepte zu untersuchen und eine Realisierung der geforderten Aspekte anzufertigen. Weiterhin sind die erstellten Lösungen in bereits existierende Teillösungen zu integrieren.

1.2 Aufgabenstellung

Ein Forschungsbereich des Fachgebietes Kommunikationsnetze beschäftigt sich mit der Untersuchung vermittlungstechnischer Prozesse in Kommunikationsnetzen. Dabei stehen vor allem paketvermittelte Netze im Vordergrund. Es wird nach neuen Wegen gesucht, die Wegewahl effizienter und an die Bedürfnisse des Nutzers angepasst zu gestalten. Damit dieses Ziel erreicht wird, soll beispielsweise auch der Kontext des Nutzers in die Routingentscheidung einfließen. Ein Konzept sieht deshalb vor, dass sich innerhalb eines Kommunikationsnetzes spezielle Router, die so genannten Kontextrouter befinden, die Informationen über dort eingebundene kontextsensitive Server enthalten. Mit diesem Wissen können die Router dann Diensteanfragen an den am besten für den Nutzer geeigneten Server weiterleiten.

Um das dargestellte Szenario umsetzen zu können, sind drei Netzkomponenten unabdingbar. Dazu gehören der Kontextrouter, der kontextsensitive Server und natürlich der Client, dessen Nutzer einen kontextsensitiven Dienst verwenden möchte. Gegenwärtig wird ein Demonstrator entwickelt, um das kontextsensitive Routing zu verifizieren. Der dort einzubindende Client soll im Rahmen dieser Diplomarbeit entwickelt werden. Als Arbeitspunkte sind daher anzugehen:

- *Auswahl der Routingsoftware*
  Der zu konzipierende Client soll für die Anfrage nach kontextsensitiven Diensten eine modifizierte Variante des AODV verwenden. Hierzu ist zu recherchieren, ob und welche bereits bestehende AODV-Implementierungen für den Client genutzt werden können. Die gewählte Lösung muss entsprechende Schnittstellen bieten, damit eine Modifikation der Funktionsweise des Protokolls erfolgen kann.

- *Konzept zur Einbindung in die Demonstratorumgebung*
  Es ist ein Konzept zu erstellen, das einen möglichst flexiblen Einsatz des Clients zulässt. Folgende Anforderungen sind hierbei zu erfüllen:

  - Umsetzung der kontextsensitiven Dienstanfrage gemäß der vorgegebenen Spezifikationen
  - Realisierung einer Schnittstelle zur manuellen Übergabe von Diensten und Kontexttypen zum Funktionstest
  - Konzeption von Anforderungen an eine Schnittstelle, über die die Anwendungen auf dem Client mit der Routingsoftware kommunizieren und dort ihre Anfrage übergeben können sowie deren Umsetzung

  - Realisierung einer Schnittstelle zur Konfiguration von Protokoll- und Funktionsparametern des Clients
  - Realisierung einer Monitoring-Schnittstelle zum Analysieren und Verfolgen der vom Client gesendeten beziehungsweise empfangenen Daten

- *Programmierung des Clients*
  Der Client ist zu realisieren und in Software umzusetzen. Diese soll unter den Betriebssystemen Windows (ab XP) und/oder Linux arbeiten.

- *Verifizieren*
  Der Client ist in den aktuellen Demonstrator zu integrieren. Mit Hilfe selbst konzipierter aussagekräftiger Testszenarios, die ausführlich zu dokumentieren sind, ist die Funktion des Clients zu verifizieren.

2 Grundlagen

Dieses Kapitel bietet einen Einblick in die Grundlagen dieser Arbeit. Hierzu wird das zugrunde liegende Nutzungsszeanrio erläutert. Anschließend findet eine Einordnung der Aufgabenstellung in das beschriebene Gesamtszenario statt. Weiterhin werden die verwendeten Netzwerkprotokolle und die dazu entwickelten Modifikationen kurz erläutert.

Grundlegende Kenntnisse zu den Themen Routing, Kontextsensitivität und Ad-hoc-Netze werden als Voraussetzung angesehen. Hierzu sei auf die folgenden Quellen verwiesen: [Pas05] und [Wen07b].

2.1 Vorstellung des Gesamtszenarios

Die vorliegende Diplomarbeit setzt sich mit der Problematik des kontextsensitiven Routing auseinander. Abbildung 2.1 zeigt einen vereinfachten Aufbau des Systems.

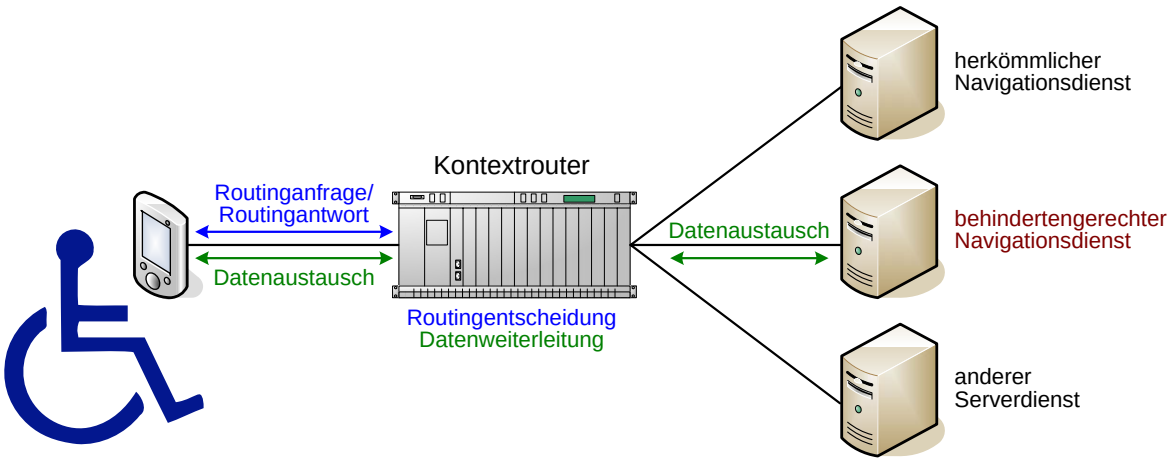

Abbildung 2.1: Beispielszenario für das kontextsensitive Routing [Wen07b]

Der Fokus dieser Ausarbeitung liegt in der Realisierung einer Software zur Unterstützung des in Abbildung 2.1 links abgebildeten Nutzers. Kapitel 2.2 erläutert die Anforderungen der zu erstellenden Software in Bezug auf das hier dargestellte Gesamtszenario.

Der mittig abgebildete Router unterscheidet sich von herkömmlichen Lösungen insofern, dass ihm kontextsensitive Informationen über die angebotenen Dienste im Netzwerk zur Verfügung stehen. Diese Informationen werden zusätzlich zur Wegewahlentscheidung herangezogen. Somit ist es möglich, Anfragen eines Teilnehmers, entsprechend der übermittelten Kontextinformationen, direkt an einen geeigneten Diensteanbieter weiterzuleiten. Weiterhin soll der in Abbildung 2.1 dargestellte Kontextrouter ebenfalls als Gateway fungieren. Hierbei wird sämtlicher Netzwerkverkehr zwischen den Diensteanbietern und den Nutzern über den Router geleitet. Dieser Kommunikationspfad ist in Abbildung 2.1 durch grüne Pfeile dargestellt.

Das Forschungsprojekt „Touristisches Assistenzsystem für barrierefreie Urlaubs-, Freizeit- und Bildungsaktivitäten“ [1] ist ein Verbundprojekt verschiedener Fachgebiete der Technischen Universität Ilmenau und der Firma „systems engineering ilmenau“. Die Idee besteht darin, Personen barrierefrei durch touristische Gebiete zu führen. Hierzu sollen Kontextinformationen der Nutzer wie Position, Orientierung und Gesundheitszustand ausgewertet werden, um an die jeweiligen Bedürfnisse des Anwenders angepasste Dienste anbieten zu können.

Somit kann das TAS-Projekt als Grundlage einer Integration von Kontext in Routingmechanismen angesehen werden.

In einem Zwischenbericht [DL04] ist der Begriff „Kontextinformation“ im Bezug auf das TAS-Projekt detailliert erläutert. Weiterhin nennt das Arichtekturkonzept [LDS05] eine Vielzahl verschiedener Kontextinformationen, die in einem derartigen Szenario Verwendung finden können. Neben persönliche Daten, wie Geschlecht, Gesundheitszustand oder Interessen sind auch technische Attribute der genutzten Hard- und Software sowie Umgebungsparameter als Beispiele für Kontextinformationen anzuführen.

Im Folgenden sei beispielsweise ein gehbehinderter Tourist im Thüringer Wald als konkreter Anwendungsfall aufgeführt. Dieser Tourist möchte nun mittels eines mobilen Endgerätes, wie zum Beispiel ein PDA oder Handy, einen Navigationsdienst in Anspruch nehmen. Hierzu übermittelt eine Clientsoftware des Nutzers die vorhandenen Kontextinformationen in einer Dienstanfrage an den in Abbildung 2.1 gezeigten Kontextrouter. Dieser ermittelt anschließend einen Dienstanbieter, der die vom Nutzer übermittelten Kontextinformationen bestmöglich unterstützt.

Es ist eine nahezu unendliche Anzahl von Einsatzgebieten vorstellbar, die Kontextinformationen zur Routensuche heranziehen können. Speziell in Verbindung mit der Wegewahl sind strukturierte Richtlinen zu erstellen, die eine Analyse und Verarbeitung von Kontextinformationen ermöglicht. [LDS05] beschreibt hierzu „kontextsensi-

[1] kurz: TAS-Projekt

tive Dienste“, die in der Lage sind, auf Kontextinformationen einzugehen.

Nach [Wen07b] kann zusammenfassend gesagt werden, das die Begriffe „Diensttyp“ und „Kontexttyp“ einen *kontextsensitiven Dienst* spezifizieren. So setzt sich ein *kontextsensitiver Dienst* aus einem definierten *Diensttyp* und mindestens einem unterstützten *Kontexttyp* zusammen.

Vor der beschriebenen Dienstanfrage durch den Nutzer müssen die in Abbildung 2.1 rechts gezeigten Dienstanbieter ihre angebotenen Dienste und unterstützten Kontexttypen dem Kontextrouter mitteilen.

Eine ausführliche Beschreibung der Kommunikationsabläufe und verwendeten Protokolle liefert [Deb07].

2.2 Einordung der Aufgabenstellung in das Gesamtszenario

Wie bereits erwähnt, beschäftigt sich die Arbeit mit der Realisierung einer Software zur Unterstützung des kontextsensitiven Routings. Wobei sich die Umsetzung auf den in Abbildung 2.1 links gezeigten Client beschränkt.

Somit muss eine mögliche Lösung unter anderem mit den bereits realisierten Komponenten kommunizieren. Hierzu wurde von [Wen07a] das Konzept einer Demonstratorumgebung für kontextsensitives Routing entworfen und in [Wen07b] umgesetzt. Der hieraus entstandene Kontextrouter bildet das Kernelement des in Kapitel 2.1 vorgestellten Gesamtszenarios. Zur Übermittlung von kontextbehafteten Dienstanfragen und -antworten kommt eine Erweiterung des Ad-hoc-Routingprotokoll AODV zum Einsatz. Der folgende Abschnitt 2.3 stellt die verwendeten Protokolle vor.

Somit ist eine Software zu erstellen, die auf Basis der beschriebenen Protokolle, Dienstanfragen des Clients ermöglicht. Nach Möglichkeit soll hierzu eine bereits existierenden AODV-Implementierung angepasst werden. Daneben sind Schnittstellen zur Übergabe der Dienst- und Kontextinformationen an die AODV-Implementierung notwendig. Weiterhin spielen Konzepte zur Nutzung der vom Kontextrouter angebotenen Dienste eine Rolle.

Am Beispiel eines gehbehinderten Touristen aus Kapitel 2.1 ist ersichtlich, dass an den Nutzer angepasste Geräte und Software benötigt werden, um die beschriebenen kontextsensitiven Dienste in Anspruch zu nehmen.

So erfordert eine mobile Nutzung die Unterstützung des verwendeten Netzwerkes zur Datenübertragung. Über dieses Netzwerk sind kontextbehaftete Dienstanfragen an den

Kontextrouter zu stellen. Weitere Schnittstellen erlauben es einer Anwendungssoftware, die erwähnten Dienstanfragen zu initiieren und deren Ergebnis entgegenzunehmen. Im Anschluss findet die eigentliche Dienstnutzung statt.

Ein dreischichtiges Modell kann hierbei die beschriebenen Vorgänge abstrahieren. Wobei die unterste Schicht die eingesetzten Protokolle repräsentiert. Die mittlere Ebene stellt eine einheitliche Schnittstelle dar. Diese soll Anwendungen einen transparenten Zugriff auf kontextbehaftete Dienste gewähren. Hierzu kommuniziert sie zum einen mit der darunter liegenden Schicht und leitet zum Beispiel Dienstanfragen ein. Zum anderen bietet sie Anwendungen einheitliche Schnittstellen zur kontextbehafteten Dienstnutzung. Daneben stellt die dritte und obere Schicht die nutzende Anwendung dar. Diese kommuniziert nach Möglichkeit ausschließlich mit der darunter liegenden mittleren Ebene, um den gewünschten kontextsensitiven Dienst zu nutzen.

Die beschriebene mittlere Schicht soll gewöhnlichen Anwendungen eine kontextbehaftete Dienstnutzung ermöglichen. Ziel ist es, die nutzenden Programme ohne spezielle Veränderungen einzusetzen.

2.3 Protokollerweiterungen zur kontextsensitiven Dienstsuche

Der folgende Abschnitt soll der Vorstellung der verwendeten Protokolle dienen. Die gezeigten Protokolle sind für die Realisierung der beschriebenen Demonstratorumgebung von Bedeutung. Ausführliche Beschreibungen der vorgestellten Protokolle sind den jeweils angegebenen Quellen sowie [Deb07] zu entnehmen.

Grundlage stellt das AODV-Routingprotokoll dar. Wobei RFC 3561 [PBRD03] das reaktive Ad-hoc-Routingprotokoll spezifiziert. Das in [Deb07] vorgestellte Konzept zur Übermittlung von kontextbehafteten Dienstanfragen durch AODV-Protokollerweiterungen wurde bereits in [Wen07b] für den Kontextrouter umgesetzt. Diese Arbeit befasst sich jedoch mit der Realisierung einer Clientsoftware zur Integration in die bestehende Demonstratorumgebung.

AODV gehört zur Gruppe der reaktiven Ad-hoc-Routingprotokolle. Somit wird es nur dann aktiv, wenn neue Routen zu bestimmen sind. Hierzu definiert [PBRD03] vier Nachrichtentypen. *RREQ*-Pakete dienen der Routensuche. Abbildung 2.2 zeigt dazu den entsprechenden Paketaufbau. *RREP*-Nachrichten werden zum einen als Reaktion auf eintreffende *RREQ*-Pakete versandt und zum anderen als so genannte *Hello-*

Nachrichten genutzt. Diese *Hello*-Nachrichten dienen der Ermittlung von erreichbaren Nachbarknoten. Abbildung 2.3 stellt den Paketaufbau einer *RREP*-Nachricht dar.

Daneben spezifiziert [PBRD03] *RERR*- und *RREP-ACK*-Nachrichten. Wobei *RERR*-Pakete versandt werden, wenn ein oder mehrere Knoten nicht mehr erreichbar sind. Eine *RREP-ACK*-Nachricht ist hingegen nur zu übermitteln, wenn ein vorheriges *RREP* mit aktivem *A*-Flag diese anfordert.

Abbildung 2.2 stellt den Aufbau einer *RREQ*-Nachricht dar, wobei entgegen RFC 3561 [PBRD03] das Flag `N` hinzugefügt wurde. Dieses Flag ist in [Deb07] beschrieben und ist in Verbindung der in Abbildung 2.5 gezeigten AODV-Protokollerweiterung gültig. Ist es auf den Wert „`1`“ gesetzt, führt der Kontextrouter kein *Rerouting* nach Erkennung eines Serverausfalls durch. Wenn das Flag jedoch nicht gesetzt ist, ermittelt der Kontextrouter einen alternativen Dienstanbieter, falls der zuvor genutzte Server ausfällt.

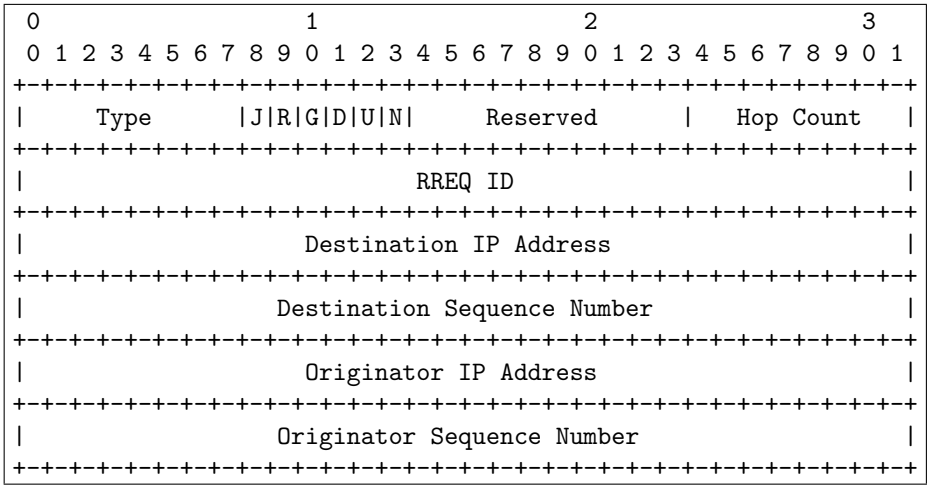

Abbildung 2.2: AODV-RREQ [PBRD03]

Die Bedeutung der in Abbildung 2.2 zu sehenden Paketfelder ist in folgender Liste zusammengefasst:

Type 1 (zur Differenzierung der verschiedenen AODV-Nachrichten)

J *Join*-Flag für Multicast-Kommunikation

R *Repair*-Flag für Multicast-Kommunikation

G *Gratious*-Flag für Unicast-RREPs (*Destination*-Feld im IP-Header enthält die Adresse des Zielknotens)

D Ist das *Destination Only*-Flag aktiv, darf nur der Zielknoten antworten

U *Unknown Sequence Number*-Flag zeigt an, dass die Sequenznummer des Zielknoten

nicht bekannt ist

N *No Reroute*-Flag (*neu*), nur in Verbindung mit der Paketerweiterung aus Abbildung 2.5 gültig, deaktiviert das *Rerouting* des Kontextrouters nach Erkennung eines Serverausfalls

Reserved ungenutzt

Hop Count Anzahl passierter Hops vom Ausgangsknoten bis zum antwortenden Knoten

RREQ ID fortlaufende Nummer, identifiziert das RREQ eindeutig

Destination IP Address IP-Adresse des gewünschten Ziels

Destination Sequence Number Sequenznummer des gewünschten Ziels

Originator IP Address IP-Adresse des Initiatorknotens

Originator Sequence Number Sequenznummer des Initiatorknotens

Abbildung 2.3 zeigt den *RREP*-Paketaufbau. Im Vergleich zur Spezifikation in RFC 3561 [PBRD03] enthält die gezeigte *RREP*-Nachricht das neu hinzugefügte Flag *No Alternatives*. Dieses neue `N`-Flag ist ebenfalls in [Deb07] detailliert beschrieben. Ist das Flag aktiv (Wert „`1`“), signalisiert der Kontextrouter einem anfragenden Client, dass keine alternativen Dienstanbieter zur Verfügung stehen.

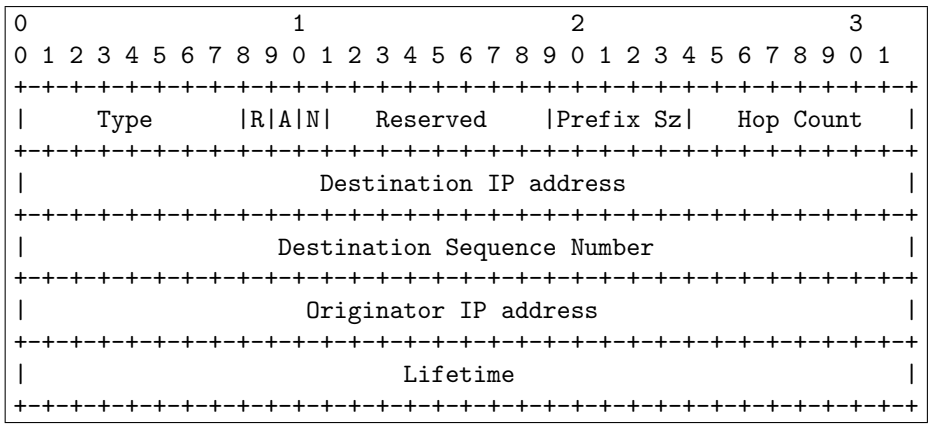

Abbildung 2.3: AODV-RREP [PBRD03]

Die Bedeutung der in Abbildung 2.3 zu sehenden Paketfelder ist in folgender Liste zusammengefasst:

Type 2

R *Repair*-Flag für Multicast-Kommunikation

A *Acknowledgement Required*-Flag erfordert eine RREP-ACK-Antwort auf das RREP-Paket

N *No Alternatives*-Flag (*neu*), nur in Verbindung mit der in Abbilung 2.6 gezeigten Erweiterung gültig, der Kontextrouter kann keinen alternativen Dienstanbieter liefern

Reserved ungenutzt

Prefix Size subnetzspezifisches Kennzeichnen von Routen

Hop Count Anzahl von Hops bis zum Empfänger

Destination IP Address IP-Adresse des gewünschten Ziels

Destination Sequence Number Sequenznummer des gewünschten Ziels

Originator IP Address IP-Adresse des Knotens, von dem das RREQ gesendet wurde

Originator Sequence Number Sequenznummer des Knotens, von dem das RREQ gesendet wurde

Auf Abbildungen der *RREP-ACK* und der *RERR*-Pakete wird im Folgenden verzichtet. Sie sind durch [PBRD03] spezifiziert und nochmals in [Wen07b] detailliert beschrieben. Weiterhin stellen diese AODV-Pakettypen keine Relevanz für die verwendeten Kontexterweiterungen und deren Kommunikationsabläufe dar.

AODV bietet nach [PBRD03] die Möglichkeit, Protokollerweiterungen an *RREQ*- und *RREP*-Nachrichten anzuhängen. Hierzu definiert RFC 3561 das in Abbildung 2.4 gezeigte generische Paketformat. Eine derartige Paketerweiterung ist direkt nach dem jeweiligen *RREQ*- oder *RREP*-Paket eingebettet. Hierbei können mehrere Erweiterungen nacheinander enthalten sein.

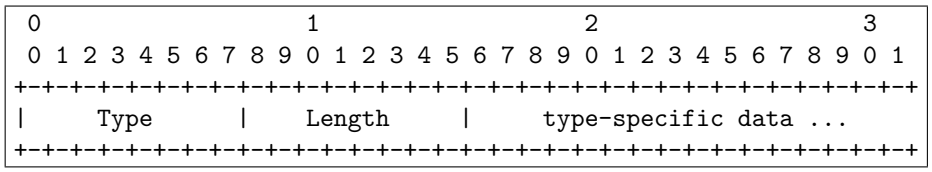

Abbildung 2.4: Prototyp der AODV-Paketerweiterungen für RREQ- und RREP-Nachrichten [PBRD03]

Beschreibungen der in Abbildung 2.4 gezeigten Paketfelder sind in folgender Liste zusammengefasst:

Type Art der Erweiterung (Wert 1-255)

Length Länge des *type-specific data*-Feldes in Byte

type-specific data Daten der Paketerweiterung

[Deb07] beschreibt weiterhin die nötigen Kommunikationsabläufe zur Nutzung der vom Kontextrouter angebotenen Dienstsuche. Hierbei können *65536* unterschiedliche

Dienste und *100* Kontexttypen angegeben werden. Während einer Dienstanfrage ist es zusätzlich möglich, die geforderten Kontexttypen mit einer Priorität zwischen *null* und *sieben* zu versehen. Wobei die geringste Priorität mit dem Wert *null* angegeben ist. Zum Übermitteln der in [Deb07] beschriebenen Dienstanfragen und -antworten kommen AODV-Paketerweiterungen sowohl in *RREQ*- als auch *RREP*-Nachrichten zum Einsatz. Abbildung 2.5 und 2.6 zeigen die jeweils verwendeten Paketerweiterungen.

Die in Abbildung 2.5 gezeigte *RREQ*-Paketerweiterung dient zur Übermittlung einer Dienstsuche von einem Client an den Kontextrouter. Hierzu ist die abgebildete Erweiterung an ein *RREQ*-Paket anzuhängen. Eine derartige *RREQ*-Nachricht mit angehängter Protokollerweiterung nach Abbildung 2.5 wird im Folgenden als *CRREQ*-Nachricht bezeichnet.

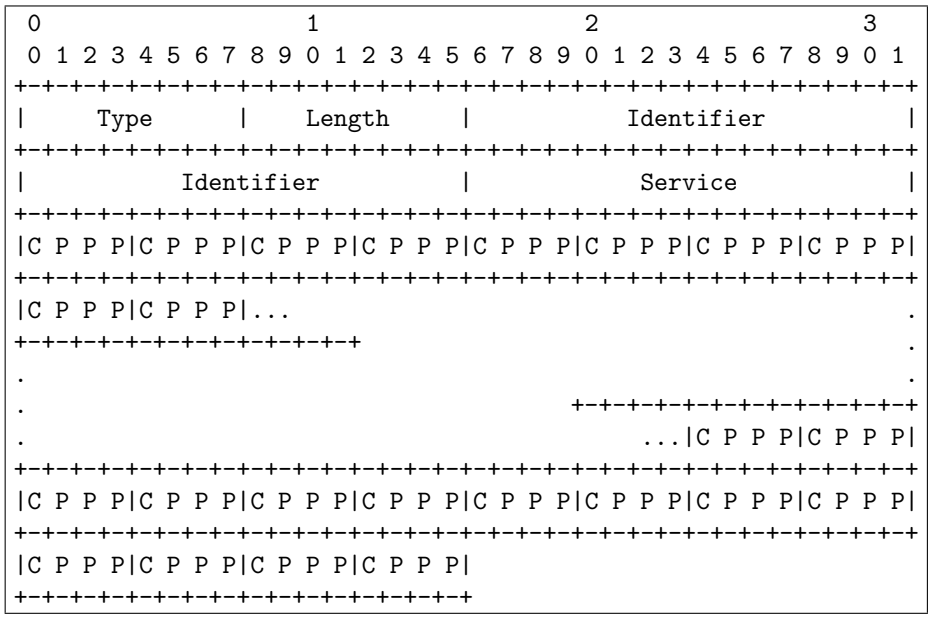

Abbildung 2.5: Erweiterung des RREQ-Headers [Wen07b]

Entgegen den Angaben in [Deb07] und [Wen07b] trägt die Paketerweiterung nach Abbildung 2.5, der im Rahmen dieser Arbeit modifizierte AODV-Implementierung, die Typnummer „16“. Der ursprünglich vorgeschlagene Wert „2“ ist bereits durch eigene Paketerweiterungen der verwendeten Implementierung vergeben. Nach Absprachen mit dem Autor des Kontextrouters Marco Wenzel nutzt dieser ebenfalls die geänderte Typnummer der Protokollerweiterung.

Die Erläuterungen der in Abbildung 2.5 gezeigten Paketfelder sind in folgender Liste zusammengefasst:

Type *16*

Length 56

Identifier Identifikator des zuletzt per CRREP empfangenen Servers wird beim ersten Senden eines RREQ auf den Wert „0“ gesetzt

Service Art des angeforderten Dienstes (Wert 0-65535)

C unterstützter Kontexttyp

P Priorität des jeweiligen Kontexttypen (Wert 0-7)

Nach dem Empfang einer *CRREQ*-Nachricht durch den Kontextrouter antwortet dieser mit Hilfe eines *RREP*-Paketes. Diesem Paket ist die in Abbildung 2.6 dargestellte Protokollerweiterung angehängt. Das Paket enthält vom Kontextrouter vergebene Informationen, die es dem Client ermöglichen, den angebotenen Dienst über den Kontextrouter zu nutzen. Hierzu müssen die IP-Pakete der jeweiligen Anwendung die in Abbildung 2.7 gezeigte IP-Option mit den erhaltenen Parametern `Session ID` und `Identifier` enthalten.

Ist ein Client jedoch mit dem angebotenen Dienst unzufrieden, kann eine erneute Dienstanfrage mittels einer *CRREQ*-Nachricht erfolgen. Dabei enthält diese den zuvor übermittelten Parameter `Identifier`. Somit kann der Kontextrouter feststellen, dass zu einer vorherigen Dienstanfrage ein alternativer Dienstanbieter zu bestimmen ist.

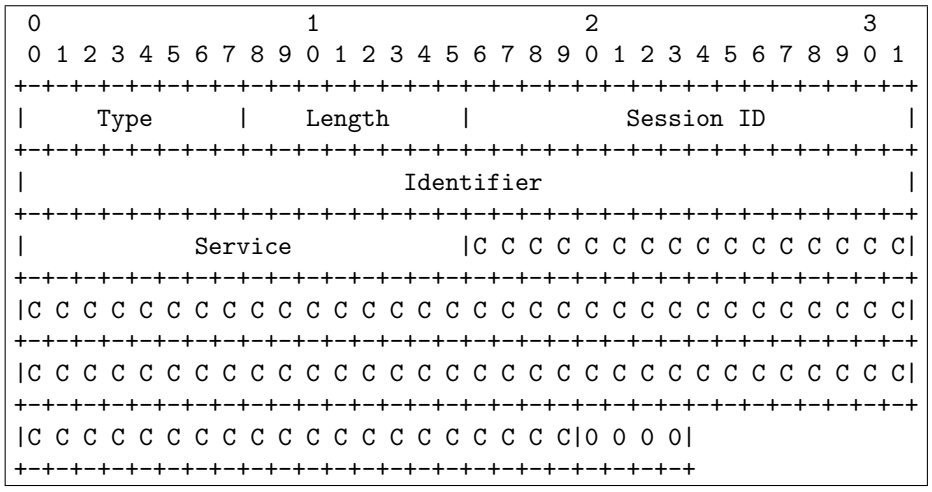

Abbildung 2.6: Erweiterung des RREP-Headers [Wen07b]

Im Folgenden werden *RREP*-Nachrichten mit der in Abbildung 2.6 gezeigten Erweiterung als *CRREP*-Pakete bezeichnet. Wie in Abbildung 2.6 weiter zu sehen ist, sind am Ende des Paketes vier Nullen eingefügt. Diese dienen lediglich zum Auffüllen des vorherigen Kontexttypbitfeldes (`C`) auf ein Vielfaches von acht Bit.

Wie in der Protokollerweiterung der RREQ-Nachrichten in Abbildung 2.5 wurde die Typenangabe entgegen dem in [Deb07] und [Wen07b] vorgeschlagenen Wert von

„2“ geändert. Der hier verwendete Wert „17“ der Typenangabe ist dem Kontextrouter ebenfalls bekannt.

Die Beschreibungen der in Abbildung 2.6 aufgeführten Paketfelder sind in folgender Liste zusammengefasst:

Type *17*

Length 21

Session ID Session-Identifikator für Rerouting

Identifier Identifikator des ausgewählten Servers

Service Art des zur Verfügung stehenden Dienstes (Wert 0-65535)

C unterstützter Kontexttyp

Wie bereits erwähnt, kann ein Client den vom Kontextrouter angebotenen Dienst über den Kontextrouter nutzen. Hierzu sind die IP-Pakete der Clientanwendung mit der in Abbildung 2.7 beschriebenen IP-Optionen zu versehen. Wobei der Client die Parameter `Session ID` und `Identifier` aus einer vorherigen Antwort auf eine Dienstanfrage erhält.

Empfängt der Kontextrouter nach einer erfolgreichen Dienstanfrage ein so markiertes IP-Paket, leitet er es an den zuvor ausgewählten Dienstanbieter weiter. Der rückläufige IP-Verkehr vom Dienstanbieter zum Client muss ebenfalls die beschriebene IP-Option enthalten, um über den Kontextrouter weitergeleitet zu werden. Hierbei kommunizieren sowohl Client als auch Dienstanbieter ausschließlich mit dem Kontextrouter. Dieses Weiterleitungsverfahren ist in [Deb07] detailliert beschrieben.

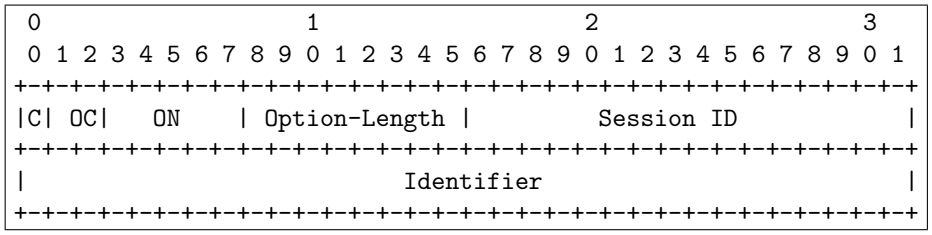

Abbildung 2.7: Erweiterung des IP-Headers [Wen07b]

Nach [Wen07b] erwartet der Kontextrouter IP-Optionen mit der *Options Number* (`ON`) `30` und gesetztem *Copied*-Flag (`C`). Dieser Wert ist laut [6] für experimentelle Anwendungen reserviert.

Die Erläuterungen der in Abbildung 2.7 gezeigten Paketfelder sind in folgender Liste aufgeführt:

C *Copied*-Flag, für den Fall der Fragmentierung soll die Option in alle Teilpakete kopiert werden

OC *Option Class* definiert die Art der Erweiterung

ON *Option Number* detaillierte Beschreibung der Art der Erweiterung

Option-Length Länge aller Options-Felder in Byte

Session ID Session-Identifikator zur eindeutigen Zuordnung einer Kommunikationsverbindung zwischen Client und Server

Identifier IP-Adresse des eigentlichen Zielknotens (Client oder Server)

Das Auffinden eines erreichbaren Kontextrouters stellt eine große Herausforderung an den Client dar. Eine mobile Nutzung der beschriebenen kontextbehafteten Dienstsuche erfordert definierte Mechanismen zum erstmaligen Kontaktieren eines Kontextrouters. [Deb07] schlägt hierfür die Verwendung von *ICMP Router Advertisement*- und *ICMP Router Solicitation*-Nachrichten vor.

Diese Nachrichten sind durch [Dee91] spezifiziert. Der Kontextrouter versendet in regelmäßigen Abständen die in Abbildung 2.8 dargestellten *ICMP Router Advertisement*-Nachrichten. Hierin ist die Kontextrouteradresse sowie deren Gültigkeitsdauer enthalten. [Deb07] schlägt weiterhin vor, den enthaltenen *Code*-Parameter auf den Wert „10“ zu setzten. RFC 1256 [Dee91] verwendet hingegen den Wert „0“. Somit sind die vom Kontextrouter erzeugten *ICMP Router Advertisement*-Nachrichten eindeutig zu identifizieren.

Empfängt ein Client eine dieser Nachrichten, ist er in der Lage einen Kontextrouter zu kontaktieren.

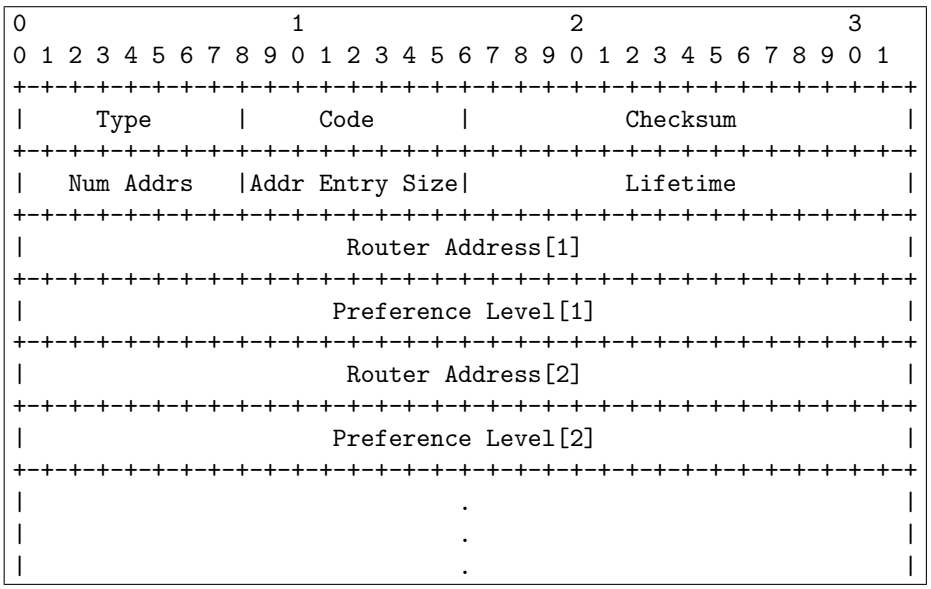

Abbildung 2.8: ICMP Router Advertisement Message [Dee91]

Eine Beschreibung der in Abbildung 2.8 zu sehenden Paketfelder ist in der folgenden Liste enthalten:

Type 9 (ICMP-Typenfeld)
Code *10* (ICMP-Codefeld)
Checksum ICMP-Prüfsumme
Num Addrs Anzahl der Adressen, die bekanntgegeben werden sollen
Addr Entry Size Anzahl der enthaltenen 32bit langen Felder
Lifetime maximale Gültigkeitsdauer der enthaltenen Informationen, in Sekunden
Router Address[n] *n = 1..Num Addrs* IP-Adressen des Routers
Preference Level[n] *n = 1..Num Addrs* zu den Adressen gehörende Präferenzgrade

Falls dem Client kein Kontextrouter bekannt ist, kann dieser durch das Versenden einer *ICMP Router Solicitation*-Nachricht einen Kontextrouter suchen. Abbildung 2.9 beschreibt den Aufbau einer solchen Nachricht. Empfängt ein Kontextrouter ein *ICMP Router Solicitation*-Paket, übermittelt er dem anfragenden Client die oben abgebildete *ICMP Router Advertisement*-Nachricht. Somit kann der Client problemlos einen Kontextrouter kontaktieren.

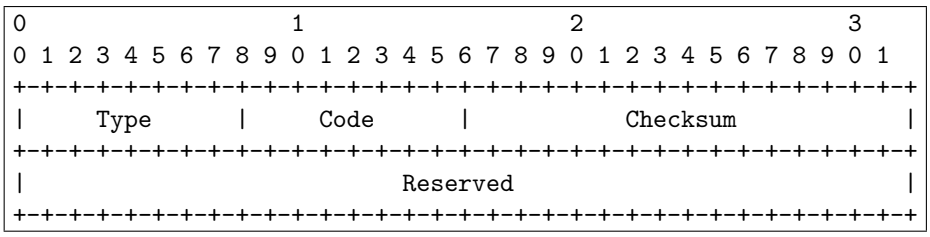

Abbildung 2.9: ICMP Router Solicitation Message [Dee91]

Wie in den *ICMP Router Advertisement*-Nachrichten aus Abbildung 2.8 schlägt [Deb07] eine Codenummer von „10“ vor. Auch für diese Nachrichten spezifiziert RFC 1256 [Dee91] den Wert „0“. Somit kann dieses Paket ebenfalls von gewöhnlichen *ICMP Router Solicitation*-Nachrichten unterschieden werden.

Die Erläuterung der in Abbildung 2.9 zu erkennenden Paketfelder ist in folgender Liste zusammengefasst:

Type 10
Code *10*
Checksum ICMP-Prüfsumme
Reserved ungenutzt

3 Auswahl der Routingsoftware

Dieses Kapitel stellt bestehende AODV Implementierungen vor und vergleicht sie miteinander. Hierbei sollen Implementierungen gefunden werden, die eine Erweiterung um kontextsensitive Fähigkeiten ermöglichen. Die notwendigen Änderungen am AODV Protokoll und das zugrunde liegende Einsatzszenario ist durch [Deb07] beschrieben. Abschließend werden in Abschnitt 3.5 die für eine Modifikation in Frage kommenden Implementierungen diskutiert.

3.1 Herkömmliche Routingarchitektur

Die Funktionsweise eines konventionellen Routers gliedert sich in die Weiterleitung *(packet forwarding)* und das Routing *(packet routing)*.

Abbildung 3.1 zeigt den schematischen Aufbau eines herkömmlichen Routers. Wie zu erkennen ist, werden die relativ simplen Weiterleitungsfunktionen (packet forwarding) innerhalb des Kernels, dem *Kernelspace*, durchgeführt. Hierzu wird für jedes eintreffende Paket anhand der Weiterleitungstabelle entschieden, über welchen Netzwerkadapter es in Richtung Ziel gesendet wird.

Die zum Teil sehr komplexe Routenberechnung und Verwaltung wird hingegen als Dienst im *Userspace* abgewickelt. Man spricht hierbei vom eigentlichen Routingdienst (Routing-Daemon). Die Komplexitität der Routenberechnung hängt vom verwendeten Routingprotokoll ab. Zu den Aufgaben des Routingdienstes gehört es ebenfalls, die Weiterleitungstabelle für den Kernel zu erstellen und zu pflegen, wobei gesonderte Schnittstellen (Kernel-APIs) genutzt werden.

Moderne Betriebssysteme nutzen diese funktionale Trennung aus vielseitigen Gründen. Über die Paketweiterleitung muss zum Beispiel für jedes eintreffende Paket entschieden werden. Daher ist diese Aufgabe effizient und schnell abzuarbeiten, was typischerweise im *Kernelspace* geschehen kann. Die zum Teil sehr rechenzeit- und speicherintensive Berechnung von Routen kann deutlich besser als Daemon im *Userspace* bewerkstelligt werden. Nicht jedes neu eintreffende Paket erfordert eine Routenberechnung. Zusätzlich würde die Komplexität der Routenberechnung die Stabilität des

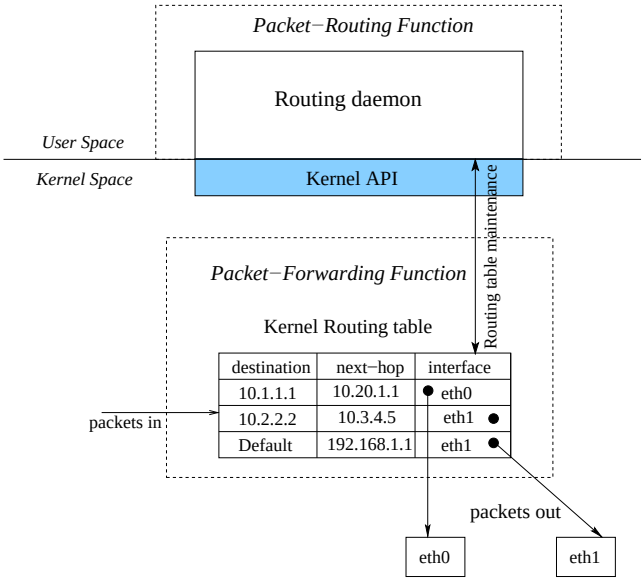

Abbildung 3.1: Herkömmliche Routingarchitektur [KZG03]

Kernels und somit des kompletten Systems unter Umständen gefährden.

Daneben erleichtert bzw. ermöglicht diese funktionale Trennung den problemlosen Austausch des verwendeten Routingprotokolls ohne den Kern des Betriebssystems anzupassen. [PD04] gibt einen ausführlichen Einblick zum Thema und geht insbesondere auf die Funktionstrennung ein.

3.2 Anforderungen des On-demand Routing

Ein Großteil der bekannten Ad-Hoc-Routingprotokolle [Wik07] lässt sich in zwei Kategorien einteilen, *proaktive* oder *reaktive* Protokolle. Proaktive oder auch *table-driven* Protokolle halten Routen zu allen möglichen Zielen lokal vor. Diese Routingtabellen werden durch periodische Kontrollnachrichten aktualisiert. Im Gegensatz dazu ermitteln reaktive oder *on-demand* Routingprotokolle nur eine Route, wenn diese benötigt wird.

Proaktive Protokolle (wie zum Beispiel DSDV [PB94]) können sich aufgrund ihrer Funktionsweise einfach als Routing-Daemon in vorhandenen Routingarchitekturen integrieren, wie es herkömmlichen Routingprotokolle (z.B. RIP, OSPF oder BGP) auch tun.

Jedoch lassen sich reaktive Routingprotokolle, wie AODV [PBRD03] oder DSR [JHM07], nicht ohne Weiteres in bestehende Routingarchitekturen integrieren. Die Anforderungen dieser Protokolle, welche neue Implementierungsmethoden notwendig

machen, werden im Folgenden erläutert und wurden aus [KZG03] entnommen.

Anforderung 1 *Behandlung ausstehender Pakete*

Im herkömmlichen Routing wird jedes Paket, welches von der Weiterleitungsfunktion bearbeitet wird, anhand der Kernel-Routingtabelle behandelt. Wenn hierbei kein Eintrag zum jeweiligen Ziel existiert, verwirft der Kernel das Paket sofort.

Dieses Verhalten ist für On-demand-Routingverfahren nicht hinzunehmen, da hier prinzipbedingt nicht alle möglichen Routen vorgehalten werden. Stattdessen sind fehlende Routen bei Bedarf (*on-demand*) zu ermitteln. Die korrekte Vorgehensweise hierbei sollte nach den folgenden Punkten erfolgen:

1. Ermitteln, ob eine neue Route gesucht werden muss.
2. Den Ad-hoc-Routing-Daemon benachrichtigen, dass einen neue Route zu finden ist.
3. Ausstehende Pakete, die auf eine Routenfindung warten, zwischenspeichern (queueing).
4. Nach erfolgreicher Routenbestimmung die wartenden Pakete absenden.

Leider existieren in modernen Betriebssystemen keine Mechanismen, die ein solches Weiterleitungsverhalten oder ein Paketzwischenspeichern im Kernel ermöglichen.

Anforderung 2 *Aktualisierung des Routencache*

Reaktive Routingprotokolle führen typischerweise eine Liste (cache) mit den zuletzt genutzten Routen im *Userspace-Routing-Daemon*, um den Protokolloverhead zu minimieren. Jeder dieser Einträge besitzt eine begrenzte Lebensdauer, die sich verlängert, wenn die entsprechende Route benutzt wurde. Bei Überschreitung der Lebensdauer verfällt ein solcher Eintrag hingegen. Anschließend ist das entsprechende Ziel sowohl aus dem Cache des Routing-Daemon als auch aus der Kernel-Routingtabelle zu löschen.

Es besteht die Notwendigkeit, dass der Routing-Daemon erfährt, ob ein Eintrag in der Kernel-Routingtabelle über eine definierte Periode ungenutzt blieb. Momentan lassen sich über vorhandene Kernel-Schnittstellen keine Informationen zur Routennutzung durch *Userspace* Programme gewinnen.

Anforderung 3 *Vermischung von Weiterleitungs- und Routingfunktionen*

Das Design einiger Ad-hoc-Routingprotokolle bietet keine strikte Trennung zwischen Weiterleitungs- und Routingfunktionen. Viele dieser Protokolle werden nur aufgrund von Datenpaketen tätig, wie zum Beispiel DSR [JHM07]. Spätestens wenn keine periodischen Aktivitäten wie Router-*advertisements*, Link-/Nachbarüberprüfung oder zeitlich begrenzte Routingeinträge vorhanden sind, muss das Routing von den Weiterleitungsfunktionen übernommen werden.

Die Implementierung dieser Protokolle in bestehende Routingarchitekturen stellt eine große Herausforderung dar. Es existieren zwei grundlegende Implementierungsansätze. Zum Einen kann das gesamte Routing innerhalb des Kernels realisiert werden, was jedoch hohe Ansprüche an die Programmierung und Wartung stellt. Andererseits kann sowohl das Routing als auch die Weiterleitung im *Userspace* realisiert werden. Hierbei ist jedes weiterzuleitende Paket vom Kernel in den *Userspace* und zurück zu transportieren, was zusätzliche Zeit in Anspruch nimmt.

Anforderung 4 *Neue Routingmodelle*

Einige Ad-hoc-Routingprotokolle nutzen unkonventionelle Routingmodelle, wie zum Beispiel *source routing* ([JM96]) oder *flow-based forwarding* ([HJ01]). Diese Routingmodelle unterscheiden sich deutlich von herkömmlichen Routingarchitekturen, so dass neue Wege zur Implementierung gefunden werden müssen.

Durch das *source routing* zum Beispiel wird der kompletten Weg eines Paketes durch die Quelle festgelegt und im Paketkopf gespeichert. Somit würde die Wegewahl nicht mehr wie bisher von jedem Router lokal, sondern lediglich vom Quellrouter durchgeführt.

Durch ein flussbasiertes Routing (*flow-based forwarding*) hingegen wird jedes Paket anhand einer ID (*flow id*) entsprechend weitergeleitet. Wobei eine ID genau ein Ziel identifiziert. Die Routingentscheidung wird hierbei von jedem Router anhand einer *flow id*-Tabelle getroffen.

Eine Implementierung in den meisten aktuellen Allzweckbetriebsystemen würde hierfür eine Modifizierung des IP-Stacks notwendig machen, um diese neuen Routingarchitekturen effizient zu unterstützen. Alternativ wären auch Kernelerweiterungen, wie zum Beispiel ladbare Module, zur Umgehung des herkömmlichen IP-Stacks vorstellbar.

Anforderung 5 *Cross-Layer-Interaktionen*

Eine Datenübertragung mittels Funk, wie es häufigerweise von Ad-hoc-Netzwerken genutzt wird, bietet zahlreiche Optimierungsansätze durch *Cross-Layer-Interaktionen*.

So nutzen einige Protokolle Parameter, wie Signalstärke oder *link status sensing*, zur Unterstützung der Wegewahl. Diese Informationen werden hierbei aus der Bitübertragung- (*physical layer*) und Sicherungsschicht (*data link layer*) gewonnen.

Eine solche Auftrennung des Schichtenkonzeptes zieht sowohl Probleme auf der konzeptionellen Ebene als auch auf der Implementierungsebene nach sich. So sind auf der Konzenptionsseite klare Richtlinien für den systematischen Zugriff auf Informationen anderer Schichten zu entwerfen. Jedoch würde ein möglicher Zugriff auf sämtliche Informationen einer benachbarten Schicht das komplette Schichtenmodell ad absurdum führen, was die bewährte Grundlage nahezu aller modernen Kommunikationseinrichtungen in Frage stellt.

Auf der Seite der Implementierung besteht eine enge Abhängigkeit zur verwendeten Hardware. So ist es vom eingesetzten Treiber abhängig, ob dieser Zugriff auf Parameter der Schicht eins oder zwei gewährt oder nicht.

3.3 Realisierung von AODV Implementierungen

Wie schon in diesem Kapitel erwähnt wurde, gehört AODV zur Klasse der *On-demand-Routingprotokolle*. Somit sollte eine AODV-Implementierung die im Abschnitt 3.2 genannten Anforderungen beachten. Da der IP-Stack moderner Betriebssysteme in der Regel für statische Netzwerke konzipiert wurde, können wichtige Ereignisse, die für den Betrieb eines reaktiven Ad-hoc-Netzwerkes notwendig sind, nicht vom Routing-Daemon ausgewertet werden. Solche Ereignisse können zum Beispiel Verbindungsabbrüche oder Paketverluste darstellen.

Um AODV in einem herkömmlichen Betriebssystem zu implementieren, muss auf die folgenden Ereignisse reagiert werden. Diese Liste deckt sich teilweise mit den in Abschnitt 3.2 genannten Anforderungen an reaktive Routingprotollimplementierungen, ist jedoch lediglich auf die Eigenschaften von AODV bezogen und wurden [CBR05] entnommen.

- *Wann wird ein RREQ initiiert?*
  Durch ein lokal erzeugtes Paket, welches an eine noch nicht bekannte Zieladresse gesendet werden soll.

- *Wann und wo sind Pakete während der Routensuche zwischenzuspeichern?*
  Während der Routensuche sollten Pakete, die an eine unbekannte Adresse gerichtet sind, zwischengespeichert (queuing) werden. Die wartenden Pakete sind zu senden, wenn Route gefunden wurde.

- *Wann wird die Lebensdauer einer aktiven Route erneuert?*
  Wenn ein Paket zu einer bekannten Destination empfangen, weitergeleitet oder gesendet wird.

- *Wann wird ein RERR für eine ungültige Route erzeugt?*
  Wenn ein Datenpaket eines anderen Knoten zu einem nicht ermittelbaren Ziel empfangen wird, muss ein RERR gesendet werden. So können die vorhergehenden Knoten und der Quellknoten das Senden über diese ungültige Route einstellen.

- *Wann soll ein RERR während eines Routing-Daemon Neustartes erzeugt werden?*
  Um Schleifenbildung zu vermeiden, sollte der AODV Routing-Daemon nach dem Neustart ein RERR an die Knoten versenden, die ihn als Router für Datenpakete verwenden wollen.

Neben diesen Punkten muss ein AODV-Knoten Verbindungsabbrüche von aktiven Routen erkennen. Tritt ein solcher Abbruch ein, wird die entsprechende Route aus der lokalen Routingtabelle entfernt. Wenn im Anschluss ein Datenpaket für eine entfallene Route eintrifft, sendet der Knoten ein RERR an den Absender. Die beiden möglichen Varianten zur Überwachung von Verbindungsabbrüchen werden im Abschnitt 3.3.2 auf Seite 25 näher erläutert.

3.3.1 Implementierungsansätze

Im Folgenden werden drei typische Ansätze für AODV-Implementierungen vorgestellt und dabei die jeweiligen Vor- und Nachteile erläutert.

Snooping

Beim snooping[1] werden wahllos alle eingehenden und ausgehenden Pakete über definierte „Abhörschnittstellen“ kopiert und Anwendungsprogrammen, wie zum Beispiel einem Routing-Daemon, zur Verfügung gestellt. Nahezu alle modernen Betriebssysteme unterstützen dieses Verfahren. Anwendungsprogramme greifen hingegen, mittels der weit verbreiteten *Packet Capture Library (libpcap)*, auf die Kernelschnittstellen zurück. Der schematische Aufbau einer solchen Implementierung ist in Abbildung 3.2 dargestellt.

[1] engl. schnüffeln

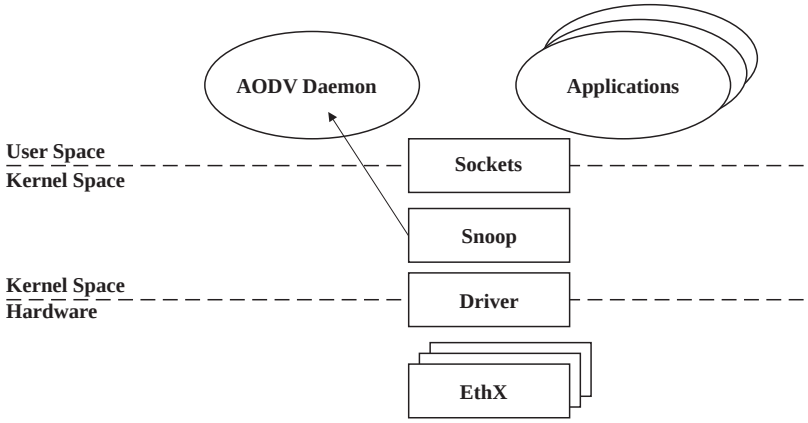

Abbildung 3.2: Implementierungsansatz: Snooping [CBR05]

Durch die definierte und breit verfügbare Schnittstelle ist eine *Pseudo-Plattformunabhängigkeit* gegeben. So können wesentliche Teile des Codes (Routing-Daemon) ohne tiefgreifende Änderungen überall dort genutzt werden, wo die libpcap-Bibliothek verfügbar ist. Ein separates Kompilieren für die jeweilige Plattform kann auch hierdurch nicht ersetzt werden.

Das Überwachen des kompletten Datenverkehrs kann nun dazu genutzt werden, die auf Seite 20 genannten Ereignisse zu erkennen und entsprechend darauf zu reagieren. So kann zum Beispiel der Routing-Daemon leicht registrieren, welche Pakete eine aktive Route nutzten und die Lebensdauer dieser entsprechend verlängern.

Der größte Vorteil dieses Lösungsansatzes ist, dass keinerlei Kernelmodifikationen notwendig sind. Daher würde sich eine solche Implementierung leicht einrichten lassen. Die zwei schwerwiegendsten Nachteile sind wiederum der zusätzliche Overhead und die Abhängigkeit von ARP. Ein ARP-Paket wird zum Beispiel dann ausgesandt, wenn die MAC-Adresse des nächsten Knoten nicht bekannt ist. Wenn nun ein ARP-request-Paket vom Routing-Daemon registriert wurde, kann man davon ausgehen, dass eine Routensuche gestartet werden muss. Wobei das versandte ARP-Paket vom lokalen Knoten erzeugt und an ein unbekanntes Ziel gerichtet ist. Da die Routensuche hierbei erst durch ein ausgehendes ARP-Paket initiiert wird, kann von zusätzlichem Overhead gesprochen werden.

Die Abhängigkeit von ARP zieht noch weitere Probleme nach sich. So ist die Synchronisierung zwischen ARP-Cache und Routingtabelle grundlegende Voraussetzung für die korrekte Funktionsweise des AODV-Protokolls. Wenn zum Beispiel der ARP-Cache einen Eintrag für eine dem Routing-Daemon unbekannte Adresse enthält, wird keine ARP-Anfrage versandt. Somit startet auch keine Routensuche. Um einen störungsfreien Ablauf des Routing zu gewährleisten, müsste die Implementierung des

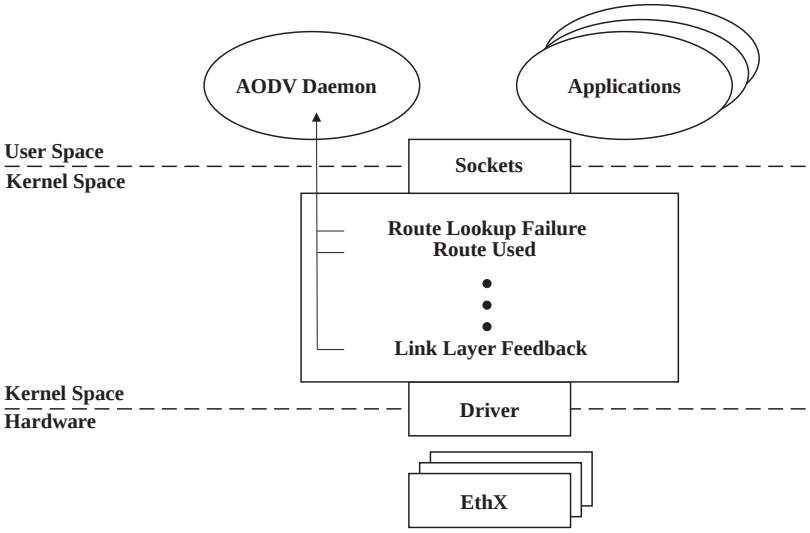

Abbildung 3.3: Implementierungsansatz: Kernelmodifikation [CBR05]

Routingprotokolls neben der Routingtabelle auch der ARP-cache überwachen und vor allem steuern können. Nur so ist gewährleistet, dass keine Diskrepanzen zwischen den beiden Tabellen auftreten und die Funktion des Routingprotokolls beeinträchtigen.

Kernelmodifikation

Eine weitere Variante auf die im Abschnitt 3.3 auf Seite 20 genannten Ereignisse zu reagieren, ist die Modifizierung des Betriebssystemkerns. Dies setzt den Zugriff auf den Kernelquellcode voraus. So besteht die Möglichkeit, misslungene Routensuchen direkt im Kernel abzufangen und anschließend ein Anwendungsprogramm (Routing-Daemon) zu benachrichtigen. Alternativ wäre es auch möglich, auf die so festgestellten Ereignisse durch eigenen Code innerhalb des Kernels zu reagieren. In Abbildung 3.3 ist der prinzipielle Aufbau eines AODV-Routers mittels Kernelmodifikation dargestellt. Hierbei ist die eigentliche Routingfunktionalität als Anwendungsprogramm realisiert.

Im Vergleich zum Snoopingansatz entsteht hierbei kein zusätzlicher Overhead, da die relevanten Ereignisse direkt erkannt werden. Die Weiterleitung der eigentlichen Datenpakete kann hier innerhalb des Kernels erfolgen. Somit müssen die Pakete nicht erst zeitaufwendig in den *Userspace* und zurück kopiert werden. Nachteilig ist jedoch die fehlende Portabilität zwischen verschiedenen Linux-Kernelversionen und die aufwändige Installation. Da hierbei die internen Kernelfunktionen modifiziert werden, um die entsprechenden Ereignisse abzufangen, ist eine Implementierung nur schwer oder gar nicht auf eine andere Kernelversion übertragbar. Diese eingeschränkte Portabilität resultiert aus den sich häufig ändernden internen Linux-Kernelschnittstellen. Weiterhin

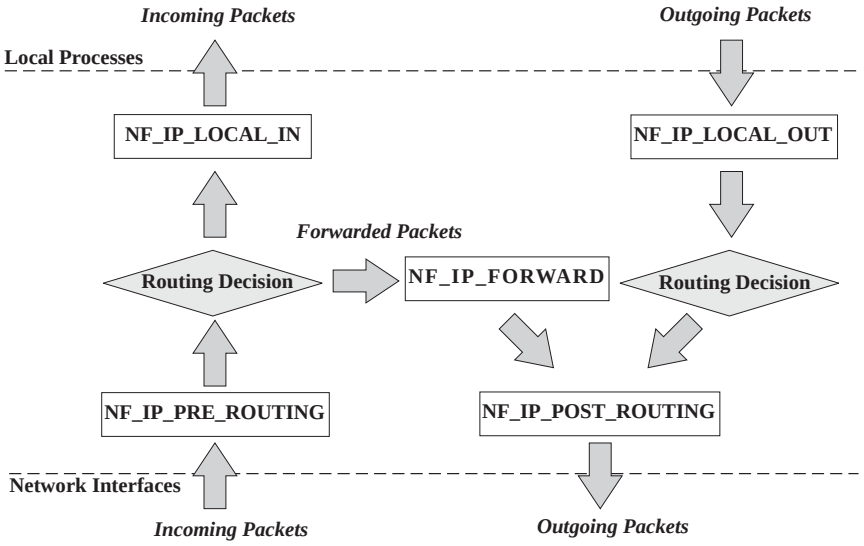

Abbildung 3.4: Linux-Netfilter Hooks [CBR05]

stellt das Einspielen von Patches zur Kernelquellcodeänderung und ein anschließendes Neukompilieren des Kernels für viele Nutzer eine unüberwindbare Aufgabe dar.

Netfilter

Das Linux-Netfilterprojekt [8] bietet eine Reihe von *hooks*[2] an verschiedenen Stellen des Linux-Protokollstacks (siehe Abbildung 3.4). So besteht die Möglichkeit, Paketströme durch eigene Filterregeln näher zu prüfen, verwerfen, modifizieren oder sogar an beliebige Anwendungsprogramme weiterzuleiten. Mit dem populären `iptables`-Programm können Anwender über diese Schnittstellen dem Kernel komplexe Filteranweisungen übergeben. Hierbei werden durch `iptables` die entsprechenden Kernelhooks den Wünschen des Nutzers entsprechend konfiguriert.

Der Netfilteransatz ist der Snoopingmethode ähnlich, jedoch benötigt dieser hierbei kein zusätzlicher Overhead durch ARP-Pakete. Abbildung 3.5 zeigt den schematischen Aufbau dieses Implementierungsansatzes. Wie zu erkennen ist, nutzt das *kaodv*-Kernelmodul die Netfilter-hooks zur Umleitung der relevanten Paketströme an den Routing-Daemon. Als relevante Paketströme können zum Beispiel alle ankommenden Pakete vom lokalen Knoten (NF_IP_LOCAL_OUT), von anderen Knoten (NF_IP_PRE_ROUTING) und alle abgehenden Pakete zu anderen Knoten (NF_IP_POST_ROUTING) eingestuft werden. Wie man in Abbildung 3.5 weiter er-

[2] engl. Haken; hier Schnittstellen zur Manipulation der IP-Paketverarbeitung

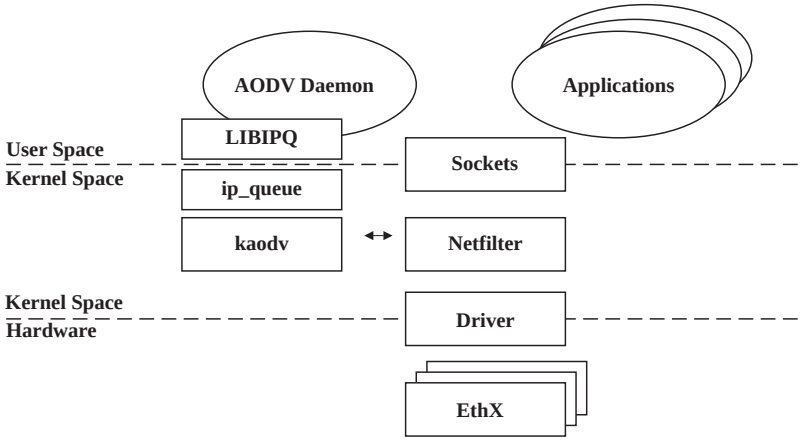

Abbildung 3.5: Implementierungsansatz: Netfilter/Kernelmodul [CBR05]

kennt, dient das *ip_queue*-Kernelmodul dem Routing-Daemon, die entsprechenden Pakete über eine Warteschlange (*queue*) weiterzuleiten. Dieser kann wiederum mittels der *libipq*-Bibliothek über die wartenden Pakete entscheiden.

Die Weiterleitung der Datenpakete kann wie in Abbildung 3.5 zu sehen ist, entweder zeitaufwendig durch den *Userspace*-Routing-Daemon oder alternativ direkt im Kernel durch das abgebildete *kaodv*-Modul erfolgen. Hierzu muss jedoch neuer Code zur Paketverwaltung im Kernelmodul implementiert werden.

Dieser Ansatz besitzt im Vergleich zu den anderen beiden Ansätzen die wenigsten Nachteile beziehungsweise Einschränkungen. So ist zum Beispiel kein zusätzlicher Overhead durch ARP-Pakete notwendig. Weiterhin lässt sich eine solche Implementierung relativ gut zwischen verschiedenen Kernelversionen portieren, da sich die Netfilterschnittstellen nur selten ändern. Die Installation ist im Vergleich zu Kernelmodifikation deutlich einfacher, weil hierbei lediglich ein Kernelmodul kompiliert werden muss und nicht der komplette Kernel. Die Abhängigkeit von einem Kernelmodul stellt hingegen auch den gewichtigsten Nachteil dieses Implementierungsansatzes dar. Jedoch wiegen die Vorteile eines dynamisch ladbaren Modules, welches nur vom Netfilterinterface abhängt, die für unerfahrene Nutzer aufwendige Installation wieder nahezu auf. So ließe sich zum Beispiel das Übersetzen und Installieren des Kernelmodules scriptgesteuert durchführen.

3.3.2 Verbindungsunterbrechungen erkennen

Neben den vorgestellten Implementierungsansätzen ist die Fähigkeit Verbindungsunterbrechungen zu benachbarten Knoten zu erkennen, essenziell für eine mögliche AODV-Realisierung. Um Bandbreite und Energie einzusparen, ist es für einen Knoten

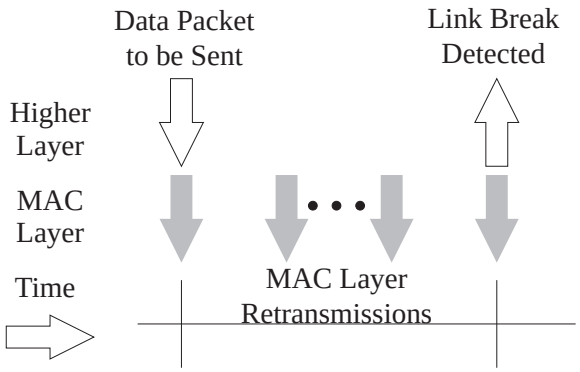

Abbildung 3.6: IEEE 802.11 Link Layer Feedback [CBR05]

hilfreich zu wissen, ob der Nachbarknoten einer Route noch in Empfangsreichweite ist und somit die Möglichkeit besteht, Datenpakete zu empfangen. Um dies zu erreichen, benötigt man eine Überwachung von Verbindungsabbrüchen, beziehungsweise eine Detektion aller erreichbaren Nachbarknoten. AODV nutzt *RERR*-Nachrichten, um die Quelle und alle dazwischen liegenden Knoten einer Route von deren Unterbrechung zu unterrichten. Verbindungsabbrüche kann AODV zum einen mittels periodischen *Hello*-Nachrichten und zum anderen durch Rückmeldungen der Sicherungsschicht (*link layer feedback*) detektieren.

Hello-Nachrichten sind periodische und im Broadcastverfahren versandte Nachrichten, die Aufschluss über die Erreichbarkeit von Nachbarknoten geben sollen. Für AODV können hierfür unter anderem sämtliche Broadcastnachrichten genutzt werden, wobei der Empfang einer solchen Nachricht eine aktive bidirektionale Verbindung indiziert. Hieraus können sich unter gewissen Umständen Probleme ergeben, da in *IEEE 802.11b*-Netzwerken Broadcastnachrichten mit einer niedrigen Bitrate und somit physikalisch weiter übertragen werden als gewöhnliche Unicastnachrichten. [LNT02] erläutert dieses Problem detailliert und gibt einige Lösungsansätze.

Wenn nach $allowed\_hello\_loss * hello\_interval$ Sekunden keine *Hello*-Nachrichten von einem Nachbarknoten empfangen werden, spricht man von einer Unterbrechung der Verbindung. Die empfohlenen Werte laut RFC 3561 [PBRD03] betragen für *allowed_hello_loss* zwei und für *hello_interval* eine Sekunde.

Im Gegensatz zu *Hello*-Nachrichten kann durch Rückmeldungen der Sicherungsschicht ein nicht mehr erreichbarer Nachbarknoten wesentlich schneller erkannt werden. Hierbei wird vorausgesetzt, dass das zugrunde liegende MAC-Protokoll (IEEE 802.11) Rückmeldungen unterstützt. Wenn der Versand eines Paketes durch das MAC-Protokoll fehlschlägt, wird im Anschluss eine Fehlermeldung an die nächst höhere Schicht gegeben. Abbildung 3.6 zeigt den prinzipiellen Ablauf einer solchen Rückmel-

dung. Somit sind höhere Schichten sofort über nicht versandte Pakete im Bilde. Im Vergleich zu periodischen *Hello*-Nachrichten ist die Latenzzeit, bis ein Übertragungsfehler vom Routing-Daemon erkannt wird, deutlich kleiner. Daher würden bei Nutzung von Rückmeldungen der Sicherungsschicht weniger Pakete verloren gehen sowie eine schnellere Routenwiederherstellung ermöglicht werden.

Mit Einführung der *Wireless Extensions* [19] ist für Linux eine einheitliche API für IEEE 802.11 basierte WLAN-Treiber und -Programme verfügbar. Somit existiert eine Abstraktionsschicht zwischen den Hardwaretreibern und Anwendungsprogrammen. Jedoch müssen die jeweiligen Treiber die entsprechenden Informationen[3] der API bereitstellen. Dies wird zum Beispiel im Linux-Kernel (Version 2.6.21) nur von den enthaltenen Treibern *HostAP*, *Aironet* und *Orinoco* unterstützt. Soweit dem Autor bekannt, ist lediglich die AODV Implementierung der Uppsala Universität [18] in der Lage, diese Rückmeldungen zur Detektion von Verbindungsunterbrechungen heranzuziehen.

Anzumerken ist jedoch, dass beide vorgestellte Varianten in *überfüllten* WLAN-Netzen an ihre Grenzen stoßen und zum Teil fehlerhafte Ergebnisse liefern können.

3.4 Vergleich bestehender AODV Implementierungen

Dieser Abschnitt dient der Gegenüberstellung der untersuchten AODV Implementierungen. Hierbei werden die wesentlichen Charakteristiken sowie Besonderheiten beschrieben. Abschließend fasst Tabelle 3.1 auf Seite 34 die relevanten Eckdaten der Implementierungen zum besseren Vergleich zusammen.

Mad-hoc stellte die erste öffentlich verfügbare AODV-Implementierung dar. Diese war komplett im Userspace realisiert und nutzte den Snoopingansatz, um die für AODV relevanten Ereignisse zu erkennen. Jedoch wies diese Implementierung einige gravierende Fehler auf. Wie im Abschnitt 3.3.1 erläutert, kann die Abhängigkeit vom ARP-Protokoll einige Probleme in der Zuverlässigkeit des Routingprotokolls hervorrufen. Weiterhin war das Zwischenspeichern (*queuing*) der Datenpakete während der Routensuche nicht fehlerfrei gelöst. Das Projekt wurde mittlerweile komplett eingestellt, so dass weder Quellcode noch Projekthomepage verfügbar sind. Somit bedarf Mad-hoc keine weitere Untersuchung im Rahmen dieser Arbeit.

[3] hier: IWEVTXDROP Event (*Packet dropped to excessive retry*)

AODV-UU

Die AODV-Implementierung der Uppsala Universität [18] nutzt den Netfilteransatz, um die für AODV notwendigen Ereignisse zu ermitteln. Hierbei kommt ein Kernelmodul zum Einsatz, welches die Nertfilter-hooks des Linux-Kernels verwendet, um den Paketstrom entsprechend zu modifizieren. Die Routingfunktionen werden hingegen im Userspace durch einen Routing-Daemon realisiert. Seit Version 0.9 nutzt das Projekt den Linux-Kernel zur Weiterleitung der Datenpakete. Somit wird der Rechen- und Zeitaufwand im Vergleich zur Weiterleitung durch Userspace-Programme deutlich minimiert. [CBR05, Abschnitt B] ermittelt hierfür einen Geschwindigkeitsvorteil um Faktor 9,7[4]. Dieser Geschwindigkeitszuwachs sollte auch bei der Nutzung aktueller Rechentechnik deutlich erkennbar sein.

Als Besonderheit kann die Portierung des Projektes in den NS-2 Netzwerksimulator [9] angesehen werden. Somit ist es möglich, den Programmcode der realen Implementierung in einer Simulationsumgebung auszuführen. Jedoch fordert diese Unterstützung mehrerer Zielplattformen ihren Tribut. Dies wird beim Betrachten des Quellcodes deutlich. Nahezu sämtliche Funktionen, die mit den jeweiligen Laufzeitumgebung (Linux Kernel oder NS-2 Simulator) kommunizieren, sind doppelt implementiert. Nur so ist es möglich, das Routingprotokoll an die deutlichen Unterschiede der jeweiligen Umgebungen anzupassen. Hierdurch vergrößert sich der Quellcode des Projektes zunehmends und wird somit unübersichtlicher sowie schlechter wartbar.

Eine Erweiterung des Projektes um kontextsensitve Routingfähigkeiten, wie sie [Deb07] beschreibt, wäre nicht zwangsläufig in der realen Implementierung *und* im NS-2 Simulator lauffähig. Diese Einschränkung resultiert auf den tiefgreifenden Änderungen am AODV-Paketformat und den damit einhergehenden Anpassungen der Schnittstellen zur Umgebung.

Neben dem im RFC 3561 [PBRD03] beschriebenen Verhalten des AODV Protokolls bietet die Implementierung der Uppsala Universität eine *Internet-Gateway Funktion* und unterstützt den Betrieb auf mehreren Netzwerkinterfaces. Mittels der Gateway Funktion kann ein Knoten (Gateway) angegeben werden, der sämtlichen *externen* IP-Verkehr tunnelt. Wie bereits im Abschnitt 3.3.2 erwähnt wurde, unterstützt AODV-UU als Einzige bekannte Implementierung *link layer feedback* zur Ermittlung von Verbindungsunterbrechungen. Somit können im Vergleich zum Abwarten von Time-outs der Hello-Nachrichten abgerissene Funkverbindungen schneller erkannt werden. Jedoch setzt dies die Unterstützung des WLAN-Treibers im Linux-Kernel voraus.

[4] Beim Einsatz von IBM Thinkpad Pentium III 1 GHz Laptops.

Eine weitere Besonderheit stellt die aktive Pflege und Weiterentwicklung des Quellcodes dar. So stammt die momentan letzte verfügbare Version (0.9.5) von 2007[5].

Die von Ian Chakeres an der University of California (Santa Barbara, USA) entwickelte ***AODV-UCSB*** Implementierung [15] basiert auf dem gleichen Design wie AODV-UU und verwendet das Kernelmodul von AODV-UU Version 0.4. Wobei sämtlicher Datenverkehr zeitaufwendig über den Routing-Daemon im Userspace geleitet wird. Das Projekt erfuhr zuletzt 2002 eine Aktualisierung und hat laut Dokumentation das Alpha-Stadium nie verlassen. Somit wird diese Implementierung im Folgenden nicht weiter zur Erweiterung um kontextsensitive Funktionen betrachtet.

AODV-UIUC

Auch AODV-UIUC [16] der University of Illinois at Urbana-Champaign nutzt den Netfilteransatz mittels eines Kernelmodules, vergleichbar mit AODV-UU. Allerdings kommt hierbei eine so genannte *Ad-hoc Support Library (ASL)* [KZG03] zum Einsatz. ASL dient als modulare Erweiterung, um sowohl reaktive als auch proaktive Ad-hoc Routingprotokolle auf Linuxsystemen zu implementieren. Die Netfilter-hooks werden hierbei vom ASL-Kernelmodul genutzt, um die für AODV relevanten Ereignisse zu bestimmen und den Userspace Routing-Daemon zu benachrichtigen. Die eigentliche Routinglogik wird wie in AODV-UU als Anwendungsprogramm (Routing-Daemon) realisiert. Die Weiterleitung der Datenpakete geschieht ebenso innerhalb des Kernels. Jedoch trennt das Projekt strikt Routing- und Weiterleitungsfunkionen voneinander. Leider unterstützt die Implementierung lediglich Linux-Kernel der 2.4'er Serie mit aktivierter Netfiltererweiterung. Somit lässt sich dieses Projekt nicht auf aktuellen 2.6'er Kerneln ausführen, die eine deutlich bessere Unterstützung für WLAN-Treiber bieten.

UoB-JAdhoc

Die in Java entwickelte UoB-JAdhoc Implementierung [21] der Universität Bremen nutzt das Snooping-Verfahren, um die für das reaktive AODV-Protokoll relevanten Informationen zu gewinnen. Als einziges der hier vorgestellten Projekte kann UoB-JAdhoc sowohl unter Windows XP und Linux ausgeführt werden. Eine konsequente objektorientierte Gliederung der Anwendung und die Nutzung von betriebssystemunabhängigen abstrakten Schnittstellen erleichtert die Unterstützung mehrerer Plattfor-

[5] siehe Tabelle 3.1

men enorm. Wie bei anderen Multiplattformimplementierungen auch, müssen jedoch die jeweiligen Schnittstellen zum Betriebssystem (Umgebung) für die entsprechenden Zielplattformen einzeln angepasst werden. [KU03] zeigt den Aufbau und Gliederung der UoB-JAdhoc-Javaimplementierung, wobei auf die Aufgaben und das Zusammenspiel der einzelnen Klassen eingegangen wird. Um Java-Programme und somit auch UoB-JAdhoc ausführen zu können, ist eine Java-Laufzeitumgebung (*Java Runtime Environment – JRE*) notwendig. Diese enthält wiederum eine *Java-VM*, welche den eigentlichen *Java-Bytecode* des Programms auf dem Zielsystem ausführt. Im Vergleich zu einem *gewöhnlichen* Programm mit identischem Funktionsumfang fordert dieses mehrschichtige Verfahren in der Regel mehr Speicher- und Rechenleistung vom auszuführenden System.

Die eigentliche Weiterleitung von Datenpaketen überlässt UoB-JAdhoc jedoch den herkömmlichen Routingfunktionen des Betriebssystems. Zur Manipulation der Kernelroutingfunktionen werden herkömmliche Shell-Befehle vom Routing-Daemon bei Bedarf abgesetzt. Zum Abhören des IP-Verkehrs kommt die Javabibliothek *Jpcap* zum Einsatz. Diese stellt eine Schnittstelle für die in Abschnitt 3.3.1 auf Seite 21 vorgestellte *libpcap*-Bibliothek zur Verfügung. UoB-JAdhoc nutzt leicht verschiedene Ansätze, um sowohl unter Linux als auch unter Windows ermitteln zu können, wann eine Routensuche initiiert werden muss. So wird zum Beispiel unter Linux die Default-Route auf das *Loopback-Interface* gesetzt. Hierdurch benötigen alle dort eintreffenden Pakete, die nicht an *127.0.0.1* gerichtet sind, eine neue Route zum jeweiligen Ziel. Unter Windows wird hingegen die MAC-Adresse des Default-Gateways auf *00:00:00:00:00:00* gesetzt. Somit benötigen alle Pakete, die von *Jpcap* übermittelt wurden und an die MAC-Adresse *00:00:00:00:00:00* gerichtet sind, eine neue Route zum gewünschten Ziel.

Weiterhin existiert ein GUI, welches unter anderem Informationen zu sämtlichen aktiven Routen darstellt. Daneben lässt sich über die graphische Oberfläche der Routing-Daemon starten und beenden sowie konfigurieren.

Laut To-Do-Liste des Projektes sind folgende Abschnitte des RFC 3561 [PBRD03] noch nicht implementiert:

- *local repair* (RFC 3561, Abschnitt 6.12)
- Subnetzrouting (RFC 3561, Abschnitt 7)
- Maßnahmen nach Neustart (RFC 3561, Abschnitt 6.13)
- Behandlung von unidirektionalen Verbindungen (RFC 3561, Abschnitt 6.8)
- *ICMP Destination unreachable*-Rückmeldung nach fehlgeschlagener Routensuche

- Paketpufferung unter Windows

Das Datum der letzten aktualisierten Version des Projektes (siehe Tabelle 3.1) lässt darauf schließen, dass die Entwicklung gestoppt wurde und somit keine neueren Veröffentlichungen folgen. Somit kann nur von einer eingeschränkten Kompatibilität zum RFC 3561 ausgegangen werden.

UoBWinAODV

Ebenfalls von der Universität Bremen wurde UoBWinAODV [20] veröffentlicht. Das in Microsoft Visual C++ entwickelte Projekt nutzt eine zum Netfilteransatz vergleichbare Methode, um die für AODV entscheidenden Informationen zu erlangen. Hierzu kommt ein so genannter „*PassThru*-Filtertreiber“ im Kernelspace zum Einsatz. Dieser *NDIS*-Treiber sitzt, schematisch gesehen, direkt über dem Netzwerkkartentreiber.

Mittels des Filtertreibers kann der Paketfluss kontrolliert und sämtliche relevanten Informationen entnommen werden. Der eigentliche Routing-Daemon wird hingegen im Userspace ausgeführt und kann mittels IOCTL-Systemaufrufen mit dem NDIS-Filtertreiber kommunizieren. Somit ist eine Verwaltung der Routingumgebung des Betriebssystems nach den AODV-Anforderungen möglich. Der hier eingesetzte NDIS-Filtertreiber basiert auf dem von Thomas F. Divine (PCAUSA, Inc.[6]) entwickelten „NDIS based Extended PassThru filter driver“, welcher durch den *Windows Driver Developers Digest*[7] veröffentlicht wurde.

Dieser Ansatz ist, wie bereits erwähnt, dem Netfilteransatz unter Linux sehr ähnlich. Auch hier kommt ein Element im Kernelspace zum Einsatz, welches durch Analyse des ein- und ausgehenden IP-Verkehrs die für AODV notwendigen Ereignisse bestimmen kann. Der eingesetzte NDIS-Filtertreiber würde im Vergleich zum Linux-Netfilteransatz dem Kernelmodul zur Nutzung der Netfilter-Hooks entsprechen.

Wie UoB-JAdhoc enthält auch UoBWinAODV ein GUI. Mittels diesem lassen sich unter anderem Informationen zu aktiven Routen abrufen sowie den Routing-Daemon starten und beenden.

Laut To-Do-Liste des Projekts sind folgende Abschnitte des RFC 3561 [PBRD03] noch nicht implementiert:

- Durchsuchen der internen Routingtabelle mittels *Longest-Prefix matching* (RFC 3561, Abschnitt 6)

[6] `http://www.pcausa.com`
[7] `http://www.wd-3.com`

- *local repair* (RFC 3561, Abschnitt 6.12)
- Subnetzrouting (RFC 3561, Abschnitt 7)
- Maßnahmen nach Neustart (RFC 3561, Abschnitt 6.13)
- Behandlung von unidirektionalen Verbindungen (RFC 3561, Abschnitt 6.8)
- *ICMP Destination unreachable*-Rückmeldung nach fehlgeschlagener Routensuche

Das Datum der letzten verfügbaren Version des Projektes (siehe Tabelle 3.1) lässt darauf schließen, dass die Entwicklung gestoppt wurde und somit keine neueren Veröffentlichungen folgen. Somit kann, wie bei UoB-JAdhoc ebenfalls, von einer eingeschränkten Kompatibilität zum RFC 3561 ausgegangen werden.

Kernel-AODV

Die von Luke Klein-Berndt am National Institute of Standards and Technology entwickelte *Kernel-AODV*-Implementierung [17] nutzt ebenfalls den Netfilteransatz zum Ermitteln der für AODV relevanten Ereignisse. Entgegen der anderen vorgestellten Implementierungen auf Netfilterbasis realisiert Kernel-AODV sämtliche Routinglogik innerhalb des Linux-Kernels als ladbares Modul. Somit ist kein weiterer Routing-Daemon notwendig. Dieser Ansatz arbeitet in Bezug auf das Pakethandling sehr leistungsfähig, da hierbei kein zeitaufwendiger Transport der Pakete in den Userspace notwendig ist. Jedoch kann ein solches Design einige Nachteile mit sich bringen. Die komplexe Protokollverarbeitung kann den Kernel unter Umständen verlangsamen, große Teile des Arbeitsspeichers belegen und sogar das komplette System zum Absturz bringen, falls in der Implementierung des Routingprotokolls ein Fehler vorhanden ist.

Daneben unterstützt das Projekt Internet-Gateway-Funktionen, arbeitete mit mehreren Netzwerkkarten zusammen, hat einfache Multicastfunktionen und bietet ein *proc file*-Interface. Mittels diesem Interface lassen sich Statistiken und Informationen zu aktiven Routen anzeigen.

Click modular router

Das *Click Modular Router Project* [1] stellt eine modulare Routingplattform zur Verfügung, welche vornehmlich zum Testen von neuen Routingfunktionen und -architekturen konzipiert wurde. Das für den Einsatz auf gewöhnlicher PC-Hardware entwickelte Open-Source Projekt ersetzt die Routingfunktionalitäten des Linux-Kernels vollständig durch ein modulares und erweiterbares Design. Der Fokus der Erweiterungen liegt in

erster Linie auf einem klar strukturierten und modularen Softwareentwurf. [KMC+00] gibt eine umfangreiche Einführung in die Struktur und Funktionsweise des Click-Routers.

Das Click-Routerprojekt bietet von Haus aus keine AODV-Funktionalitäten. Marco Wenzel untersuchte in [Wen07a] existierende Möglichkeiten zur Integration des AODV-Routingprotokolls. Hierbei fand sich eine bestehende Erweiterung auf Basis der Masterarbeit [Bra05] von Bart Braem. Der Quellcode dieses Click-Elements ist leider nicht frei verfügbar. In [Wen07a] heißt es hierzu: „Jene Quelltexte wurden von Bart Braem nur unter der Voraussetzung weitergegeben, diese nicht zu veröffentlichen und nicht weiter zu verbreiten."

Der Click-Router kann sowohl im Kernelspace als auch um Userspace ausgeführt werden. Hierzu sind nahezu alle Module (*Elemente*) doppelt implementiert. Der Betrieb im Userspace ist als Test- und Debuggumgebung vorgesehen, während die Nutzung im Kernelspace für den Produktiveinsatz gedacht ist.

Die in [Bra05] beschriebene AODV-Implementierung bietet ebenfalls diesen Dualismus. Jedoch stellt [Wen07a] fest, dass der im Kernel lauffähige Click-AODV-Teil an Funktionen des *Network Simulator – NS-2* [9] gebunden ist. So gelang es dem Autor Marco Wenzel nicht, die NS-2 Abhängigkeiten der Kernel-Implementierung ohne Beeinträchtigung der AODV-Funktionalitäten zu entfernen. Somit wurde auf die Implementierung im Userlevel zurückgegriffen.

Nach [Bra05, Kapitel 5.3] ist die Click-AODV-Implementierung kompatibel zum RFC 3561 [PBRD03]. Einige Einschränkungen existieren jedoch. Es werden weder Multicastadressierung noch unidirektionalen Verbindungen unterstützt. Weiterhin kann nur ein Netzwerkinterface genutzt werden.

Der Implementierungsansatz unterscheidet sich deutlich von den bisher vorgestellten Projekten. Das AODV-Element im Userspace nutzt das so genannte TUN/TAP-Interface, um alle Pakete vom Kernel zu empfangen beziehungsweise zu senden. Hierbei handelt es sich um ein virtuelles Netzwerkinterface. Alle Pakete, die ein TUN/TAP-Gerät entgegen nimmt, werden an ein Programm im Userspace weitergeleitet und umgekehrt.

Im Rahmen der Diplomarbeit [Wen07b] wurde auf der Basis dieser Software der in Abschnitt 2.1 beschriebene Kontextrouter implementiert. Hierzu wurden die in Abschnitt 2.3 beschriebenen Kontextprotokollerweitungen genutzt.

	AUDV-UU	AODV-UIUC	UoB-JAdhoc	UoBWin AODV	Kernel-AODV	Click-AODV
Implementierungsansatz	Netfilter (Kernelmodul)	Netfilter (Kernelmodul)	snooping	wie Netfilter*	Netfilter (Kernelmodul)	TUN/TAP-Interface*
Routinglogik	Userspace	Userspace	Userspace	Userspace	Kernelspace	Userspace
Weiterleitung von Datenpaketen	Kernelspace	Kernelspace	Kernelspace	Kernelspace	Kernelspace	Userspace
RFC konform	ja	nein (Draft ver. 10)	ja*	ja*	ja	ja
Erkennung von Verbindungsunterbrechunen	Hello-Nachrichten, Link layer feedback	Hello-Nachrichten	Hello-Nachrichten	Hello-Nachrichten	Hello-Nachrichten	Hello-Nachrichten
Plattform	Linux (Kernel 2.4, 2.6), ARM/Mips+	Linux (Kernel 2.4)	Java (Windows/Linux)	Windows XP	Linux (Kernel 2.4)	Linux (Kernel 2.4, 2.6)
letzte Aktualisierung	23.07.2007 (Ver. 0.9.5)	30.07.2004	12.4.2005	21.4.2005	15.04.2004	2005

*) siehe Text
+) ARM/Mips-Prozessor basierte Geräte wie iPAQ, Zaurus und WRT54G Router

Tabelle 3.1: Vergleich der AODV-Implementierungen

3.5 Diskussion

Eine Wahl bezüglich einer zu modifizierenden Implementierung soll vornehmlich von folgenden Aspekten beeinflusst werden:

- Funktionalität (RFC kompatibel)
- Eignung für Produktiveinsatz
  - Anzahl unterstützter Plattformen
  - aktive Projektpflege
- Qualität/Dokumentation des Quellcodes

Beim Betrachten der Tabelle 3.1 wird ersichtlich, dass lediglich *AODV-UIUC* nicht RFC-konform ist. Wie in Abschnitt 3.4 beschrieben, besitzen *UoB-JAdhoc* und *UoB-WinAODV* der Universität Bremen diesbezüglich weitere Einschränkungen. Diese geben zum Beispiel keine *ICMP destination unreachable*-Rückmeldungen nach einer fehlgeschlagenen Routensuche. Hierbei kann ein gewöhnliches Anwendungsprogramm nicht direkt erkennen, ob die zu kontaktierende IP-Adresse nicht erreichbar ist.

Das Kriterium „Eignung für den Produktiveinsatz“ bezieht sich auf Nutzungsmöglichkeiten des im Kapitel 2.1 vorgestellten Einsatzszenarios. Hierbei ist es wünschenswert, wenn mehrere Hardwareplattformen unterstüzt werden. Durch eine aktive Projektpflege, beziehungsweise regelmäßige Aktualisierungen, ist die Wahrscheinlichkeit größer, dass neuere Hardware oder Betriebssysteme unterstützt werden.

Unter diesen Gesichtspunkten stechen zwei Implemetierungen besonders hervor. *UoB-JAdhoc* ist durch die Nutzung der Programmiersprache Java sowohl in einer Windows- als auch in einer Linuxumgebung lauffähig. Jedoch wird das Projekt nicht mehr aktiv weiterentwickelt, so dass zum Beispiel nur der 2.4'er Linuxkernel unterstützt wird. *AODV-UU* ist die einzige Implementierung, die noch aktiv weiterentwickelt und gepflegt wird. Laut Dokumentation lässt sich das Projekt auf ARM- und Mips-Prozessoren basierenden „Linux-Handheldrechnern“ betreiben.

Die Qualität beziehungsweise eine Dokumentation des Quellcodes beeinflussen die Einarbeitungs- und Bearbeitungszeit enorm. Somit kann diesem Aspekt eine gesonderte Rolle zugewiesen werden, da die praktische Modifikation einer AODV-Implementierung einer der Kernbestandteile der Aufgabenstellung darstellt. Auch hier heben sich wieder zwei Projekte von den Vorgestellten ab. So sind im Quellcode der *AODV-UU*-Implementierung zahlreiche Kommentare enthalten, die eine Einarbeitung deutlich erleichtern. Weiterhin ist der Quellcode gut strukturiert, was durch eine funktionale

Quellcodeaufteilung in verschiedenen Dateien unterstützt wird. So sind zum Beispiel die Funktionen zum Verarbeiten von RREP- oder RREQ-Nachrichten in separaten Datei abgelegt. Zusätzlich existieren mehrere Veröffentlichungen, die unter anderem den Aufbau des *AODV-UU*-Projektes beschreiben. Hierfür können [CBR05] und [KZG03] als Beispiel angeführt werden.

Das *Click Modular Router Project* zeichnet sich als zweite Implementierung durch einen hochgradig modularen und klar gegliederten Aufbau aus. Jedoch ist der rechtliche Aspekt bezüglich der Weitergabe und Verbreitung des AODV-Quellcodes hinderlich. Weiterhin kann man auf der Projekthomepage [1] neben zahlreichen Veröffentlichungen auch eine umfangreiche Onlinedokumentation aller in *click* enthaltenen Elemente abrufen. Die Dokumentation der click-AODV-Elemente ist lediglich der Masterarbeit [Bra05] zu entnehmen.

Nach Abwägung der beschriebenen Eigenschaften der vorgestellten Projekte kristallisierten sich zwei potentielle Implementierungen zur Modifikation heraus. Einerseits überzeugt *AODV-UU* durch eine stabile Codebasis mit zahlreichen Funktionen und noch aktiver Projektpflege. Andererseits sticht der *click modular Router* durch seinen streng strukturierten Aufbau hervor. Jedoch ist die AODV-Implementierung hier nur eine zusätzlich hinzugefügte Erweiterung ohne frei verfügbaren Quellcode. Wie bereits erwähnt, hat Marco Wenzel im Rahmen seiner Diplomarbeit [Wen07b] das AODV-Element um die in Kapitel 2.3 vorgestellten Kontextprotokollerweitungen ergänzt. Diese Modifikationen wurden genutzt, um den in Abschnitt 2.1 beschriebenen Kontextrouter zu implementieren.

Der ganzheitliche Ansatz, das Routing des Linuxkernels komplett zu ersetzen, kann weitere Probleme beziehungsweise Nebenwirkungen nach sich ziehen. So stellte sich beim Testen der von Marco Wenzel zur Verfügung gestellten Implementierungen heraus, dass zum Beispiel die häufig genutzten Paketfilter des Linuxkernels nicht mehr funktionierten.

Daneben hat die rechtliche Situation des AODV-Quellcodes den Autor dazu bewogen, **AODV-UU** als Basis für den praktischen Teil dieser Arbeit zu verwenden.

4 Modifikation der AODV-UU Implementierung

Dieses Kapitel dient der Beschreibung des praktischen Teil der Aufgabenstellung. Hierbei wurden die in Kapitel 2.3 vorgestellten Protokollerweiterungen in AODV-UU (Version 0.9.5) integriert. Somit kann ein Betrieb als *Kontextclient*, im Rahmen des in Abschnitt 2.1 vorgestellten Szenarios gewährleistet werden.

Weiterhin wird ein Überblick der internen Struktur der AODV-UU-Implementierung gegeben. Dies soll als Grundlage der beschriebenen Ansätze zur Modifikation des AODV-Quellcodes dienen. Zur Nutzung der neu eingeführten Funktionen bietet Abschnitt 4.3 eine Dokumentation der Anwendungsschnittstelle. Diese Schnittstelle ermöglicht es speziellen kontextsensitiven Anwendungen unter anderem die in Kapitel 2.1 beschriebenen Kontextrouter zu kontaktieren. Erst hierdurch kann eine kontextbehaftete Dienstsuche und anschließende Nutzung erfolgen.

4.1 Beschreibung des internen Aufbaus

Wie bereits erwähnt, lässt sich AODV-UU zum einen als eigenständiger Routing-Daemon für reale Anwendungen betreiben. Zum anderen besteht die Möglichkeit, die Implementierung als Simulation im Netzwerksimulator *NS-2* [9] auszuführen. Die Umschaltung dieser Betriebsmodi erfolgt durch C-Präprozessoranweisungen während des Kompilierens.

Eine derartige duale Verwendungsmöglichkeit der Implementierung bedarf zahlreicher Fallunterscheidungen im Quellcode. Fundamental unterschiedliche Schnittstellen zum umgebenden System sowie grundlegende Unterschiede in der Funktionsweise stellen hierbei die größte Herausforderung dar. Somit sind nahezu alle Funktionen, die mit dem umgebenden System kommunizieren, doppelt implementiert oder enthalten die erwähnten Fallunterscheidungen. Eine Anpassung des Paketformates und der Kommunikationsabläufe nach Abschnitt 2.3 ließe sich nur in beiden Umgebungen ausführen, wenn auch hier die genannten Besonderheiten der Origialimplementierung umgesetzt

werden.

AODV-UU ist in der Programmiersprache *C* geschrieben und nutzt einen prozeduralen Ansatz. Lediglich die Quellcodeteile zur Integration in den Netzwerksimulator *NS-2* ergänzen die objektorientierte Struktur des Simulatorquellcodes.

Die Datei `main.c` bildet die zentrale Komponente der Implementierung. Hier werden unter anderem globale Variablen definiert, Kommandozeilenargumente ausgewertet und sämtliche Programmelemente initialisiert. Weiterhin enthält die Funktion `main` die Hauptschleife des Programms. Innerhalb dieser Schleife wird durch einen `select`-Aufruf ermittelt, welcher Programmteil neue Daten oder Informationen zu verarbeiten hat.

```
int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
    timeval *timeout);
```

Quellcode 4.1: `select` Syntax

Die `select`-Funktion wird von der GNU C Library (kurz: glibc) [4] bereit gestellt und dient in erster Linie der synchronen I/O-Multiplexsteuerung. Quellcode 4.1 zeigt den Syntax der Funktion. Für die Nutzung in AODV-UU sind folgende Parameter der `select`-Funktion relevant: `nfds`, `*readfds` und `*timeout`.

`*readfds` enthält die Menge der zu überwachenden Socket- oder Dateideskriptoren. Nach dem Funktionsaufruf sind der Variable alle zum Lesen bereitstehenden Deskriptoren zu entnehmen. Mittels `nfds` wird der höchste numerische Wert der zu überwachenden Datei- beziehungsweise Socketdeskriptoren übergeben. `*timeout` definiert eine maximale Zeitspanne, die die Funktion verstreichen lassen kann, bevor sie zurückkehrt. Der Rückgabewert von `select` enthält die Anzahl der Deskriptoren, die ihren Status geändert haben.

Somit ist es leicht möglich, mehrere geöffnete Dateien oder Sockets zu überwachen und auf Änderungen, wie zum Beispiel dem Eintreffen von neuen Daten oder Nachrichten, zu reagieren. Nach dem Funktionsaufruf kann mittels dem Makro `FD_ISSET` (siehe Quellcode 4.2) festgestellt werden, welcher der überwachten Deskriptoren seinen Status geändert hat.

```
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Quellcode 4.2: Syntaxen der FD_ Makros, zur Verwaltung von `fd_set`-Variablen

```
typedef void (*callback_func_t) (int);

#define CALLBACK_FUNCS 5
static struct callback {
    int fd;
    callback_func_t func;
} callbacks[CALLBACK_FUNCS];
```

Quellcode 4.3: `callbacks` Array Definition, aus Datei `defs.h` und `main.c`

```
int attach_callback_func(int fd, callback_func_t func)
```

Quellcode 4.4: `attach_callback_func` Funktionskopf, aus Datei `main.c`

Die in Quellcode 4.2 abgebildeten Makros dienen unter anderem der Initialisierung (`FD_ZERO`) der `fd_set`-Variable. Daneben kann mittels `FD_SET` ein neuer File- oder Socketdeskriptor hinzugefügt werden. `FD_CLR` löscht hingegen einen Dateideskriptor aus der `fd_set`-Variable.

Eine Zuordnung zwischen Socketdeskriptor und jeweiliger Behandlungsfunktion realisiert AODV-UU über das Array `callbacks` (siehe Quellcode 4.3). Die bereits erwähnten Initialisierungsfunktionen, welche nach dem Programmstart aufgerufen werden, registrieren ihre Behandlungsfunktionen für eintreffende Nachrichten oder Daten mit Hilfe der Funktion `attach_callback_func` aus der Datei `main.c` (siehe Quellcode 4.4. Diese füllt das Array `callbacks` mit dem übergebenden Deskriptor `fd` und einem Zeiger auf die Behandlungsfunktion `func`.

Registrierung der callback-Funktionen

Wie der Typendefinition aus Quellcode 4.3 zu entnehmen ist, sind alle hinterlegten Behandlungsfunktionen vom Typ `callback_func_t`. Somit dürfen sie keinen Rückgabewert liefern (`void` Argument) und besitzen genau einen Parameter vom Typ `int`. Die oben besprochene Funktion `attach_callback_func` wird von folgenden Intialisierungsfunktionen aufgerufen:

- `aodv_socket_int()`, aus Datei `aodv_socket.c`
  Hier wird ein UDP-Socket zum Empfangen und Senden von AODV-Paketen im System registriert. Als Behandlungsfunktion dient `aodv_socket_read`.

- `nl_init()`, aus Datei `nl.c`
  Das AODV-UU-Kernelmodul `kaodv` erkennt die für den AODV-Daemon relevan-

ten Ereignisse und übermittelt sie via des hier erzeugten Sockets in den Userspace an die Behandlungsfunktion `nl_kaodv_callback`. Auf diese Weise wird dem AODV-Daemon unter anderem die Notwendigkeit einer Routensuche signalisiert.
Weiterhin richtet die Funktion `nl_init()` einen Socket zum Hinzufügen und Entfernen von Einträgen der Kernelroutingtabelle ein. Hierzu wird die Funktion `nl_rt_callback` registriert.

- `llf_init()`, aus Datei `llf.c`
  Mit dem Kommandozeilenargument „`-f`“ kann AODV-UU das im Kapitel 3.3.2 besprochene *link layer feedback* nutzen, um Verbindungsabbrüche der Nachbarknoten zu detektieren. Dieses optionale Feature kann nur genutzt werden, wenn der verwendete WLAN-Treiber die entsprechenden Statusmeldungen der Sicherungsschicht zur Verfügung stellt. Ist dies der Fall, wird ein Socket und `llf_callback` als Behandlungsfunktion der *link layer feedback*-Meldungen registriert.

Nach einem `select`-Aufruf wird festgestellt, welcher der registrierten Sockets zum Lesen bereit ist, beziehungsweise neue Nachrichten liefert. Somit kann die jeweilige Behandlungsfunktion aufgerufen werden und die anstehenden Daten aus dem Socket lesen. Anschließend werden die empfangenen Nachrichten oder Datenpakete analysiert und entsprechend reagiert.

Wenn zum Beispiel ein neues AODV-RREP-Paket im System eintrifft, wird dies von der Funktion `aodv_socket_read` gelesen und an die Funktion `aodv_socket_process_packet` übermittelt. Dort wird im Anschluss der AODV-Pakettyp ermittelt und die jeweilige Behandlungsfunktion aufgerufen. Im Falle eines AODV-RREP-Paketes ist die Funktion `rrep_process` aus der Datei `aodv_rrep.c` zuständig.

Auf diesem Schema fußt die gesamte Bearbeitung aller eintreffenden Ereignisse. So ist es leicht möglich, den Weg zur Bearbeitung eines Ereignisses zu verfolgen.

Time-out-Behandlung

Die Bearbeitung von *Time-outs* erfolgt nach einem etwas anderem Verfahren als die oben vorgestellte Behandlung von Ereignissen durch *Callback*-Funktionen. Vor jedem `select`-Aufruf in der Hauptschleife wird die in Quellcode 4.5 abgebildete Funktion `timer_age_queue` aufgerufen.

Diese Funktion wird von der Datei `timeout_queue.c` bereit gestellt und arbeitet alle bereits abgelaufenen Time-ous ab. Weiterhin liefert sie als Rückgabewert die Zeit-

```
timeout = timer_age_queue();

if ((n = select(nfds, &rfds, NULL, NULL, timeout)) < 0) {
...
```

Quellcode 4.5: Aufruf der Time-out-Bearbeitung und anschließendes `select`, aus Datei `main.c`

```
typedef void (*timeout_func_t) (void *);
struct timer {
        list_t l;
        int used;
        struct timeval timeout;
        timeout_func_t handler;
        void *data;
};

int NS_CLASS timer_init(struct timer *t, timeout_func_t f, void *data);
void NS_CLASS timer_set_timeout(struct timer *t, long msec);
int NS_CLASS timer_remove(struct timer *t);
```

Quellcode 4.6: Definition und Funktionsköpfe der relevanten Time-out-Behandlungsfunktionen, aus Datei `timer_queue.h`

spanne bis zum Verstreichen des nächsten Time-outs. Dieser Rückgabewert wird, wie in Quellcode 4.5 zusehen ist, als Parameter für den anschließenden `select`-Aufruf in der Hauptschleife verwendet. So ist sichergestellt, dass `select` rechtzeitig zurückkehrt und somit im Anschluss auf die abgelaufenen Time-outs reagieren kann.

Die Verwaltung von Time-outs erfolgt mittels einer verketteten Liste. AODV-UU stellt hierfür unter anderem die in Quellcode 4.6 abgebildeten Funktionen und Datenstrukturen bereit. Wie man sieht, ist für jeden zu registrierenden Timer eine Bearbeitungsfunktion vom Typ `timeout_func_t` anzugeben. Diese wird in einem Datensatz vom Typ `struct timer` im Element `handler` gespeichert. Nach Ablauf eines Timers erfolgt die Ausführung der hinterlegten Funktion. Die Dateien `timer_queue.h` und `timer_queue.c` stellen die abgebildeten Datenstrukturen und Funktionen bereit.

Die in Quellcode 4.6 dargestellten drei Funktionsköpfe dienen einerseits zum Erzeugen eines neuen Timers (`timer_init`), andererseits dem Definieren eines Time-out-Zeitpunktes (`timer_set_timeout`) sowie dem Entfernen eines bestehenden Timers (`timer_remove`).

4.2 Implementierungsansätze der Protokollerweiterungen

Basierend auf den im vorherigen Kapitel 4.1 beschriebenen Aufbau und Funktionsweisen von AODV-UU kristallisieren sich zwei wesentliche Aufgabenfelder einer Modifizierung heraus. Zum einen sind die Datenstrukturen der neuen AODV-Paketerweiterungen umzusetzen und zum anderen Funktionen zur Reaktion auf neue Ereignisse zu erstellen. Abschnitt 4.2.1 dieses Kapitels geht im Detail auf die bereits vorhandenen und neu zu erstellenden Datenstrukturen sowie die genutzten AODV-UU-Funktionen ein.

Zum anderen ist eine Schnittstelle zu erzeugen, die es Anwendungen erlaubt, kontextbehaftete Dienstanfragen stellen zu können. Abschließend ist diese Schnittstelle funktional mit den neuen Funktionen der kontextsensitiven Routen- beziehungsweise Dienstsuchen zu verknüpfen. So muss es gewährleistet sein, dass nach einer Dienstanfrage über die neue Anwendungsschnittstelle ein passendes *AODV-CRREQ*-Paket abgesetzt wird. Nach Ankunft einer entsprechenden Antwort eines Kontextrouters muss die Nachricht der anfragenden Anwendungen zugeordnet werden und schließlich über die Schnittstelle eine Meldung ausgegeben werden. Abschnitt 4.2.2 beschreibt Ansätze der realisierten Schnittstelle zur Kommunikation mit kontextsensitiven Anwendungen.

Die in diesem Kapitel vorgestellten AODV-UU-Modifizierungen sollen folgenden selbst gestellten Anforderungen gerecht werden:

- Nutzung bereits vorhandener Strukturen/Architekturen und Funktionen
- Schaffung eines möglichst flexiblen und einfachen zu nutzenden Interfaces
- Gleichzeitige Nutzung der Schnittstelle durch mehrere Anwendungen

4.2.1 Datenstrukturen und bestehende Funktionen

AODV-UU bringt bereits von Haus aus einen Basisdatentyp `AODV_ext` (siehe Quellcode 4.7) für Protokollerweiterungen mit. Wie in Quellcode 4.8 zu sehen ist, existieren mit den Funktionen `rreq_add_ext` und `rrep_add_ext` Mechanismen zum Hinzufügen von beliebigen AODV-Protokollerweiterungen. Daneben sind die Funktionen `rrep_process` und `rreq_process` aus den Dateien `aodv_rrep.c` und `aodv_rreq.c` bereits auf die Erkennung und Verarbeitung von Protokollerweiterungen vorbereitet.

```
/* An generic AODV extensions header */
typedef struct {
    u_int8_t type;
    u_int8_t length;
    /* Type specific data follows here */
} AODV_ext;
```

Quellcode 4.7: Generische Datenstruktur für AODV-Protokollerweiterungen, aus Datei `defs.h`

```
AODV_ext *rreq_add_ext(RREQ * rreq, int type, unsigned int offset, int len, char
    *data);
AODV_ext *rrep_add_ext(RREP * rrep, int type, unsigned int offset, int len, char
    *data);
```

Quellcode 4.8: AODV-UU Funktionen zum Hinzufügen von AODV-Protokollerweiterung, aus Datei `aodv_rreq.c` und `aodv_rrep.c`

Diese Funktionen werden nach dem Eintreffen eines *RREP* beziehungsweise *RREQ*-Paketes aufgerufen und verarbeiten diese.

Die in [Deb07] vorgeschlagenen Typnummer „2“ der neuen Kontextprotokollerweiterung ist durch AODV-UU bereits vergeben. Quellcode 4.9 zeigt die bereits vorhandenen Erweiterungsnummern sowie die neu definierten Typnummern. Diese Änderung am Protokoll wurde in Absprache mit dem Autor des Kontextrouters Marco Wenzel vollzogen. Somit kann eine problemlose Kommunikation zwischen Kontextclient und -router gewährleistet werden. Die Nutzung der *extension type*-Nummern „16“ und „17“ bietet noch genügend Abstand zu den bereits genutzten und zukünftigen Paketerweiterungen, so dass eine störungsfreie Nutzung möglichst lange gewährleistet werden kann.

Neben der Anpassung von vorhandenen Funktionen und Datenstrukturen ist das eigentliche Paketformat, der in Kapitel 2.3 beschriebenen Kontexterweiterungen, um-

```
/* AODV Extension types */
#define RREQ_EXT 1
#define RREP_EXT 1
#define RREP_HELLO_INTERVAL_EXT 2
#define RREP_HELLO_NEIGHBOR_SET_EXT 3
#define RREP_INET_DEST_EXT 4

/* Karsten Renhak -- 17-09-2007 */
/* add new extension types */
#define RREQ_CONT_EXT 16
#define RREP_CONT_EXT 17
```

Quellcode 4.9: Definition der Typnummern der AODV-UU-Protokollerweiterungen, aus Datei `defs.h`

```
#define MAX_CONT_TYPES 100
#define ARRAY_SIZE(A,B) ((A % B) ? (A / B + 1) : (A / B))

typedef struct {
#if defined(__LITTLE_ENDIAN)
        u_int8_t p_2:3;
        u_int8_t c_2:1;
        u_int8_t p_1:3;
        u_int8_t c_1:1;
#elif defined(__BIG_ENDIAN)
        u_int8_t c_1:1;          /* context type flag */
        u_int8_t p_1:3;          /* context type priority */
        u_int8_t c_2:1;          /* context type flag */
        u_int8_t p_2:3;          /* context type priority */
#else
#error "Adjust your <bits/endian.h> defines"
#endif
} RREQ_CONTEXT_DEF;

typedef struct {
        u_int32_t id;            /* store it always in network byte order! */
        u_int16_t service;       /* service type number */
        RREQ_CONTEXT_DEF ct[ARRAY_SIZE(MAX_CONT_TYPES,2)];       /* array of context
            type flags and priorities */
} C_RREQ_EXT;

typedef struct {
        u_int8_t c_1:1; /* context type flag  */
        u_int8_t c_2:1;
        u_int8_t c_3:1;
        u_int8_t c_4:1;
        u_int8_t c_5:1;
        u_int8_t c_6:1;
        u_int8_t c_7:1;
        u_int8_t c_8:1;
} RREP_CONTEXT_DEF;

typedef struct {
        u_int16_t session;
        u_int32_t id;            /* store it always in network byte order please! */
        u_int16_t service;       /* service type number */
        RREP_CONTEXT_DEF ct[ARRAY_SIZE(MAX_CONT_TYPES,8)];       /*  array of context
            type flags */
} C_RREP_EXT;
```

Quellcode 4.10: Definition der Datenstrukturen für RREQ- und RREP-Protokollerweiterung, aus Datei `cont_ext.h`

zusetzen. Quellcode 4.10 zeigt die hierfür vorgenommenen Anpassungen zur Definition des beschriebenen Paketformates. Zu beachten ist, dass sämtliche neuen Funktionen und nahezu alle mit den Erweiterungen zusammenhängenden Datenstrukturen in seperaten Dateien abgelegt wurden. Dies dient in erster Linie der Wartbarkeit des Quellcodes. Zusätzlich sind so fast alle Änderungen von der ursprünglichen AODV-UU-Implementierung abgrenzbar. Um die restlichen Modifikationen an den von AODV-UU bereitgestellten Dateien erkennbar zu gestalten, wurden diese durch aussagekräftige Kommentare im Quellcode gekennzeichnet.

4.2.2 Realisierungsansätze der Anwendungsschnittstelle

Aus den Zielstellungen der AODV-UU-Anpassung im Abschnitt 4.2 auf Seite 42 geht hervor, dass eine Schnittstelle, über die Anwendungen kontextbezogene Dienstanfragen stellen können, möglichst einfach aufgebaut und zu nutzen sein soll. Daher wurde ein gewöhnlicher *TCP-Socket* als Grundlage für diese Schnittstelle gewählt. Dies hat den Vorteil, dass bereits existierende Funktionen zum Empfangen und Senden von Nachrichten genutzt werden können. Auch auf der Seite der nutzenden Anwendung können die vom Betriebssystem bereitgestellten Funktionen zur Socket Kommunikation genutzt werden. Abbildung 4.1 zeigt einen typischen Verlauf der Client/Serverkommunikation bei Nutzung von TCP-Sockets.

Wie zu erkennen ist, wartet der Serverprozess nach einem `listen()`-Aufruf auf eingehende Verbindungen. Nach dem Eintreffen einer TCP-Verbindungsanfrage kreiert der Serverprozess einen neuen Socketdeskriptor, über den die eigentliche Kommunikation mit dem Client abgewickelt wird. Anschließend können weitere Verbindungswünsche bearbeitet werden. Somit ist es dem Serverprozess möglich, mehrere Clientverbindungen gleichzeitig zu erzeugen, sofern der Entwickler des Programms dies vorsieht.

Aus Abbildung 4.1 wird ersichtlich, dass jeder verbundene Client über einen eigenen Socketdeskriptor identifizierbar ist. Durch Verbindungsauf- und -abbau jedes Clients muss der Serverprozess, in diesem Fall AODV-UU, die zu überwachenden Socketdeskriptoren dynamisch verwalten können. Jedoch ist es AODV-UU mittels der Funktion `attach_callback_func` nur möglich, neue Socketdeskriptoren vor dem Betreten der Hauptschleife zum Überwachen zu registrieren.

Dieses Manko wurde durch das Ändern der Funktion `attach_callback_func` sowie dem Hinzufügen von `detach_callback_func` in der Datei `main.c` beseitigt. Somit kann ein Socketdeskriptor mit entsprechender *Callbackfunktion* im bereits erwähnten `callbacks`-Array hinzugefügt und entfernt werden. Damit diese Änderungen sich auf

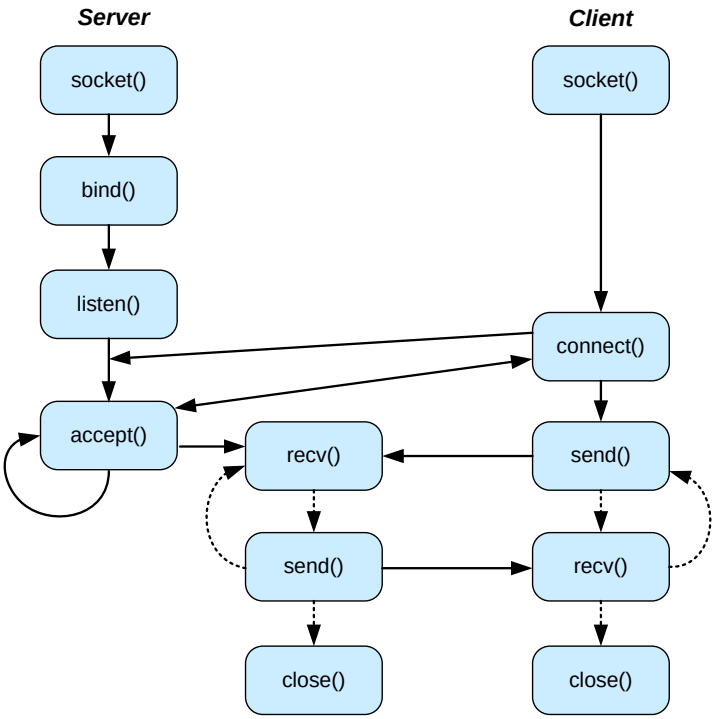

Abbildung 4.1: Typischer Ablauf einer TCP-Socketverbindung

den `select`-Aufruf innerhalb der Hauptschleife auswirken, ist neben dem `callbacks`-Array auch die genutzte `fd_set`-Variable des `select`-Aufrufes zu aktualisieren.

Ein ungültiger beziehungsweise nicht genutzter Eintrag im `callbacks`-Array ist nun durch den Deskriptorwert von „`-1`“ und einem Zeiger auf die Behandlungsfunktion mit dem Wert „`NULL`“ zu erkennen. Daneben ist durch die Funktionen `attach_callback_func` (Quellcode 4.11) und `detach_callback_func` (Quellcode 4.12) neben der Verwaltung des `callbacks`-Arrays auch die `fd_set`-Variable für den `select`-Aufruf zu aktualisieren. Hierzu sind die Makros `FD_SET` beziehungsweise `FD_CLR` (siehe Quellcode 4.2) zu verwenden.

Wie in Quellcode 4.13 zu erkennen ist, wird die globale `fd_set`-Variable `readers` zur Übergabe der registrierten Socketdeskriptoren genutzt. Sie wird in jedem Durchlauf der Hauptschleife erneut eingelesen sowie von den Funktionen `attach_callback_func` und `detach_callback_func` entsprechend verändert, was aus Quellcode 4.11 und 4.12 hervorgeht.

Somit wurden alle Voraussetzungen geschaffen, einen TCP-Serversocket anzulegen sowie mehrere Clientverbindungen verwalten zu können. Vergleichbar mit den im Abschnitt 4.1 auf Seite 39 erwähnten Initialisierungsfunktionen wird nun die Funktion

```
int attach_callback_func(int fd, callback_func_t func)
{
        int i;

        if (nr_callbacks >= CALLBACK_FUNCS) {
                fprintf(stderr, "callback/socket attach limit reached!!\n");
                exit(-1);
        }
        for( i = 0; i < CALLBACK_FUNCS; i++) {
                if (callbacks[i].fd < 0) {
                        callbacks[i].fd = fd;
                        callbacks[i].func = func;
                        nr_callbacks++;

                        FD_SET(callbacks[i].fd, &readers);
                        if (callbacks[i].fd >= nfds)
                                nfds = callbacks[i].fd + 1;
                        break;
                }
        }
        return 0;
}
```

Quellcode 4.11: Modifizierte Funktion `attach_callback_func`, aus Datei `main.c`

```
int detach_callback_func(int fd, callback_func_t func) {
        int i;

        if (nr_callbacks < 1) {
                fprintf(stderr, "no callback function or socket left to remove!!\n");
                exit(-1);
        }
        for( i = 0; i < CALLBACK_FUNCS; i++) {
                if (callbacks[i].fd == fd && callbacks[i].func == func) {
                        callbacks[i].fd = -1;
                        callbacks[i].func = NULL;
                        nr_callbacks--;

                        FD_CLR(fd, &readers);
                        break;
                }
        }
        return 0;
}
```

Quellcode 4.12: Neue Funktion `detach_callback_func`, aus Datei `main.c`

```
while (1) {
        memcpy((char *) &rfds, (char *) &readers, sizeof(rfds));
        timeout = timer_age_queue();

        if ((n = select(nfds, &rfds, NULL, NULL, timeout)) < 0) {
            if (errno != EINTR)
                alog(LOG_WARNING, errno, __FUNCTION__,
                     "Failed select (main loop)");
            continue;
        }

        if (n > 0) {
                for (i = 0; i < CALLBACK_FUNCS; i++) {
                        if (callbacks[i].fd > 0 && callbacks[i].func != NULL) {
                                if (FD_ISSET(callbacks[i].fd, &rfds)) {
                                        (*callbacks[i].func) (callbacks[i].fd);
                                }
                        }
                }
        }
}                               /* Main loop */
```

Quellcode 4.13: Modifizierte Hauptschleife, aus Datei `main.c`

`aodv_cont_ext_init` aus der Datei `cont_ext.c` während der Programminitialisierung aufgerufen. Diese Funktion erzeugt unter anderem den TCP-Serversocket und registriert mittels `attach_callback_func` die Funktion `handle_new_cont_ext_app_sock` aus der Datei `cont_ext.c` als Behandlungsfunktion für neue Clientverbindungen. Mittels der C-Präprozessordefinition „`#define CONT_EXT_IF_PORT 22222`“ aus Datei `cont_ext.h` wird der Port für eingehende Verbindungen des Serversockets definiert.

Die Funktion `handle_new_cont_ext_app_sock` erhält nach einem `accept()`-Aufruf den neuen Socketdeskriptor der Clientverbindung und registriert ihn ebenfalls mit Hilfe von `attach_callback_func` mit der Behandlungsfunktion `handle_cont_ext_app_msg`. Somit können ab dem nächsten Durchlauf der Hauptschleife Nachrichten von verbundenen Clients, in diesem Fall Kontextanwendungen, von der Funktion `handle_cont_ext_app_msg` gelesen werden.

Auf der Grundlage der im Abschnitt 4.2 auf Seite 42 genannten Zielstellungen wurde ein textbasiertes Interface zur Übergabe der kontextbehafteten Dienstanfragen sowie zum Verändern von Parametern implementiert. Eine Übermittlung von Befehlen und Parametern mittels gewöhnlichen ASCII-Zeichen erleichtert das Testen und Debuggen enorm, da Nachrichten einfach „von Hand“ an das Interface gesendet und gelesen werden können. Hierzu ist lediglich ein simples Programm, welches auf Nutzereingaben wartet und diese dann über den TCP-Socket an das Interface sendet und Antworten am Bildschirm ausgibt, notwendig.

Auf der anderen Seite, dem Server-Socket, ist jedoch ein deutlich höherer Aufwand zu

```
char *strchr(const char *s, int c);
char *strtok_r(char *str, const char *delim, char **saveptr);
int strcasecmp(const char *s1, const char *s2);
```

Quellcode 4.14: Verwendete Funktionen zum parsen der eintreffenden Nachrichten, in Datei `con_ext.c`

treiben, um die empfangenen Nachrichten zu analysieren und gegebenenfalls zu reagieren. Die Funktion `handle_cont_ext_app_msg` nutzt die in Quellcode 4.14 abgebildeten Funktionen zum Analysieren beziehungsweise Parsen der eintreffenden Nachrichten.

Ausschlaggebend hierfür sind verschachtelte Aufrufe der Funktion `strtok_r`. Sie dient zum Zerlegen von Strings anhand von Trennzeichen (Delimiter), welche über das `const char *delim`-Argument spezifiziert werden. Am Beispiel einer typischen Nachricht zur Dienstsuche „`context-request service=145 ct=1,0:24,7`“ wird im Folgenden der Ansatz zum Analysieren der eintreffenden Nachrichten gezeigt.

Im ersten Schritt wird die Zeichenkette nach Freizeichen-Delimiter zerlegt. Im Anschluss überprüft ein `strcasecmp`-Aufruf, ob der Teilstring ein gültiges Schlüsselwort beziehungsweise ein Befehl repräsentiert. Wenn dies nicht zutrifft, wird mit Hilfe von `strchr` nach Vorkommen von weiteren Trennzeichen gesucht. Ist die Suche erfolgreich, wird ein weiterer `strtok_r`-Aufruf auf dem Teilstring ausgeführt. Wobei dieser Substring anhand der gefundenen Trennzeichen zerlegt wird.

Der beschriebene Prozess wird so lange wiederholt, bis in den zerlegten Teilzeichenketten keine gültigen Schlüsselwörter oder Trennzeichen auffindbar sind. Abbildung 4.2 zeigt die Struktur der `strtok_r`-Aufrufe, um die oben aufgeführten Beispielnachricht zu analysieren. Eine Übersicht zu allen implementierten Kommandos, Syntaxen und deren Bedeutung gibt Anhang A.3.

Erkennt die Funktion `handle_cont_ext_app_msg` alle notwendigen Argumente eines Befehls, wird die entsprechende Reaktionsfunktion mit den empfangenen und aufbereiteten Argumenten aufgerufen. So initiiert zum Beispiel die Funktion `c_rreq_route_discovery` das Versenden eines *CRREQ*-Paketes zur kontextbehafteten Dienstsuche. Eine vollständige Übersicht der erstellten und modifizierten Funktionen und deren Bedeutung ist Anhang A.1 und A.2 zu entnehmen.

4.3 Anwendungsschnittstelle

Entgegen den Ausführungen im Kapitel 4.2.2 befasst sich dieser Abschnitt mit der Nutzung der realisierten Schnittstelle. Diese Schnittstelle ermöglicht, kontextbehaftete

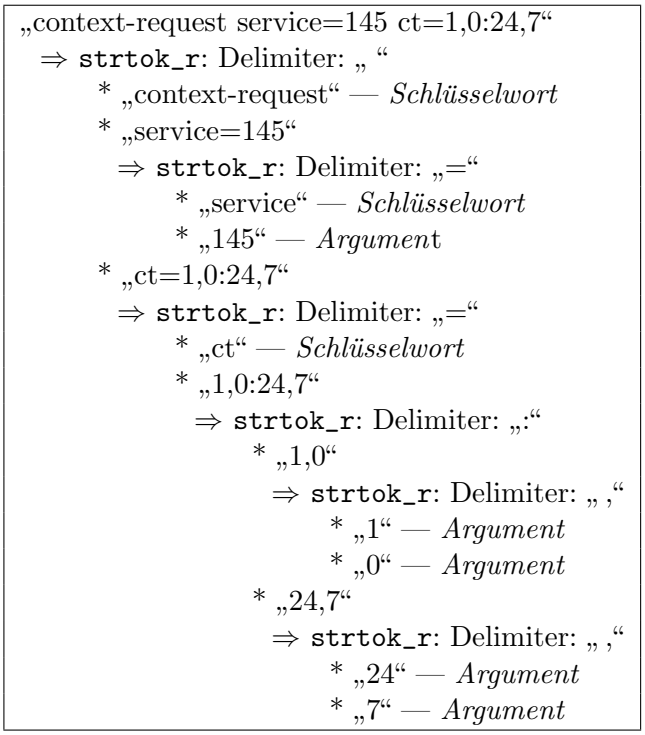

Abbildung 4.2: Hierarchie der `strtok_r`-Aufrufe zum Stringparsen

Dienstanfragen dem Routing-Daemon zu übergeben sowie einige Parameter zu modifizieren. Anhang A.3 führt alle implementierten Befehle und deren Syntax auf. Im Anschluss ist die erstellte Software zum Testen der Schnittstelle Gegenstand des Kapitels 4.3.2. Abschließend wird im Kapitel 4.3.3 ein Konzept eines deutlich flexibleren Interfaces zur Nutzung kontextsensitiver Dienste präsentiert und mit der realisierten Schnittstelle verglichen.

Abbildung 4.3 illustriert den Aufbau und das Zusammenspiel der vorgestellten Komponenten zur Suche und Nutzung kontextbehafteter Dienste. Die Box *Kontext-Anwendungsschnittstelle* rechts oben stellt das realisierte Dienstprogramm `cont_client` zur Nutzung des in diesem Kapitel beschriebenen Interfaces dar. Kapitel 4.3.2 beschreibt diesen Testclient genauer.

Der rosa Pfeil im oberen Bereich der Abbildung 4.3 hingegen symbolisiert die notwendigen Nutzereingaben zur Übermittlung von Nachrichten an den Routing-Daemon. Hierbei wartet `cont_client` auf Nutzereingaben und leitet sie ohne vorherige Prüfung über den TCP-Socket an den Routing-Daemon weiter. Die abgebildeten dünnen rosa Pfeile repräsentieren diesen Teil der Kommunikation.

Nach dem Absetzen eines Befehls über die neue Schnittstelle zur Kontextdienstsuche

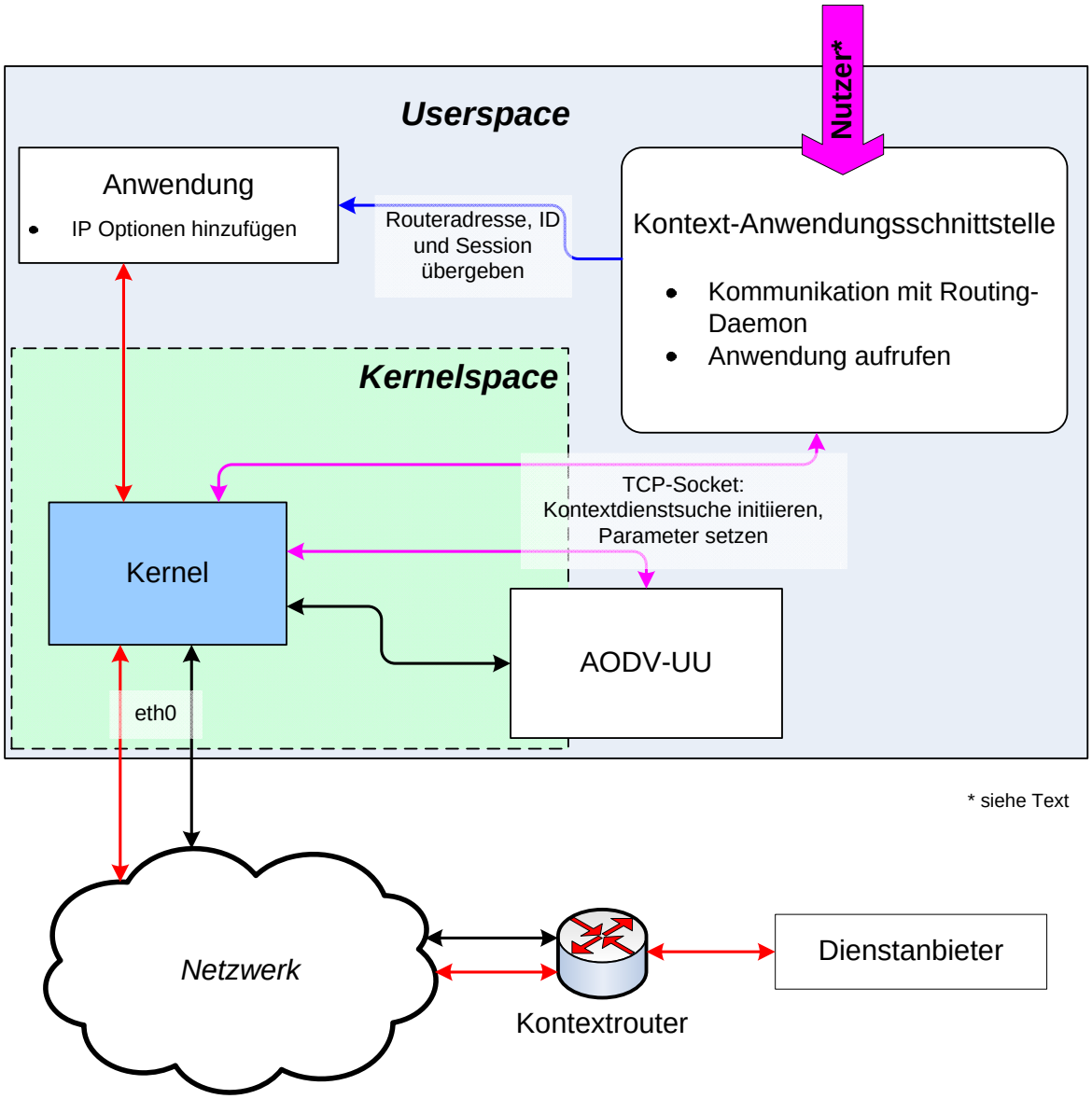

Abbildung 4.3: Schema des realisierten Interface zur Suche kontextsensitiver Dienste

kommuniziert AODV-UU mit einem erreichbaren Kontextrouter, was durch die schwarzen Pfeile symbolisiert wird. Die IP-Adresse des Kontextrouters ermittelt der Routing-Daemon entweder aus den periodisch versandten *ICMP-Router-Advertisement*-Nachrichten oder vom Benutzer selbst. Ist AODV-UU jedoch keine Kontextrouteradresse bekannt oder der gewünschte Router nicht erreichbar, wird eine *ICMP-Router-Solicitation*-Nachricht versendet, um eine Antwort eines Kontextrouters zu provozieren. Der Aufbau dieser ICMP-Pakete ist durch RFC 1256 [Dee91] spezifiziert und in Kapitel 2.3 beschrieben.

Über den TCP-Socket erhält die Kontext-Awendungsschnittstelle entweder eine posi-

tive oder negative Antwort vom Routing-Daemon. Im Fall einer erfolgreichen Anfrage wird neben der IP-Adresse des Kontextrouters auch eine *ID*- und *Session*-Nummer übermittelt. Der in Abbildung 4.3 dargestellte blaue Pfeil symbolisiert diesen Programmaufruf.

Diese drei Parameter werden im Anschluss genutzt, um eine Kontextanwendung aufzurufen. Hierzu wurde das Programm `ping` aus dem *Gentoo*-Paket `net-misc/iputils` (Version 20070202) dahingehend modifiziert, dass durch einen neuen Kommandozeilenparameter *ID* und *Session* übergeben werden können.

Somit ist es dem modifizierten `ping`-Programm (`CIPPING`) möglich, die in [Deb07] und Kapitel 2.3 beschriebenen IP-Optionen den *ICMP*-Echo-Request-Paketen hinzuzufügen und an den Kontextrouter zu senden. Dem Router ist es anhand der gesetzten IP-Optionen möglich, die eintreffenden Pakete an den jeweiligen Dienstanbieter weiter zu leiten. Dieser Kommunikationsweg wird durch die abgebildeten roten Pfeile repräsentiert.

Wie durch Abbildung 4.3 erkennbar ist, hat die Anwendung dafür Sorge zu tragen, den Dienstanbieter beziehungsweise den Kontextrouter zu kontaktieren und gegebenenfalls IP-Optionen einzufügen. Somit muss die Anwendung Kenntnisse über die genutzten Protokolle besitzen und daneben Möglichkeiten bieten, die vom Kontextrouter übermittelten *ID* und *Session*-Parameter entgegen nehmen zu können.

Abschnitt 4.3.3 hingegen stellt ein alternatives Konzept zur Nutzung kontextsensitiver Dienste vor. Hierbei liegt der Fokus, im Gegensatz zur implementierten Schnittstelle, auf der Portabilität und Unabhängigkeit der nutzenden Anwendungen. Somit soll gewährleistet werden, dass Anwendungen kontextbehaftete Dienste möglichst ohne Kenntnisse der verwendeten Protokolle und Strukturen nutzen können.

4.3.1 Schnittstellensyntax

Wie bereits erwähnt, stellt die modifizierte AODV-UU-Implementierung einen TCP-Socket zur Übergabe von kontextbehafteten Dienstanfragen sowie zur Modifizierung einiger Parameter zu Verfügung. Die Kommunikation zwischen Anwendung[1] und dem AODV-Daemon erfolgt textbasiert und unterscheidet nicht zwischen Groß- und Kleinschreibung. Sämtliche empfangenen Befehle, beziehungsweise Kommandos, der Anwendung zum AODV-Daemon werden von diesem bestätigt.

Auf Grund der in Abbildung 4.2 verdeutlichten hierarchischen Analyse und Zerlegung eintreffender Nachrichten sind bestimmte Trennzeichen notwendig, um einzelne

[1] hier: Programm `cont_client`

Befehle sowie deren Argumente voneinander abzugrenzen. Hierbei werden die Befehle durch *Leerzeichen* voneinander getrennt.

Generell kann zwischen Schlüsselworten mit und ohne zusätzlichen Argumenten unterschieden werden. Hierbei trennt das „=“-Zeichen den Befehl und das zugehörige Argument voneinander ab. Im Fall des `ct`-Schlüsselwortes ist, wie in Abbildung 4.2 zeigt, eine weitere Zerlegung des Argumentes notwendig.

Tablle 4.1 zeigt alle realisierten Schlüsselwöter der implementierten Anwendungsschnittstelle. Eine Ausführliche Beschreibung der einzelnen Befehle ist Anhang A.3 zu entnehmen.

Schlüsselwort	**Beschreibung**
context-request	leitet eine kontextbehaftete Dienstanfrage ein
no-reroute	setzt das *No Reroute*-Flag in allen zukünftigen CRREQ-Paketen
reroute	deaktiviert das *No Reroute*-Flag in allen zukünftigen CRREQ-Paketen
blacklist-print	gibt alle IP-Adressen der Serverblackliste aus
crreq-timeout-print	gibt die Zeitspanne aus, die verstreichen muss, bis ein CRREQ-Paket von einer Time-Out-Funktion erneut versendet wird
force-soli	erzwingt das Senden einer *ICMP-Router-Solicitation*-Nachricht
help	gibt eine Liste aller verfügbaren Kommandos aus
quit	beendet die Verbindung zum TCP-Socket des Routing-Daemons
service	spezifiziert den gewünschten Diensttyp
ct	spezifiziert die gewünschten Kontexttypen sowie deren Prioritäten
router	übergibt die IP-Adresse eines Kontextrouters
id	spezifiziert den *ID*-Wert des CRREQ-Paketes
blacklist-add	fügt eine IP-Adresse der Serverblackliste hinzu
blacklist-del	entfernt eine IP-Adresse aus der Serverblackliste
crreq-timeout	setzt die Time-Out-Zeitspanne der CRREQ-Pakete

Tabelle 4.1: Schlüsselwörter der Anwendungsschnittstelle

4.3.2 Vorstellung Testclient

Zur Nutzung der im Abschnitt 4.3 beschriebenen Schnittstelle bedarf es eines separaten Programms, welches unter anderem die vom Nutzer eingegebenen Nachrichten an den Routing-Daemon sendet und dessen Antworten ausgibt. Dieses Programm ist in Abbildung 4.3 als *Kontext-Anwendungsschnittstelle* bezeichnet und ruft nach einer erfolgreichen Antwort auf eine Dienstanfrage ein modifiziertes `ping`-Programm auf. Die Quellen des angepassten `ping`-Programms basierten auf dem *Gentoo*-Paket `net-misc/iputils`, Version 20070202.

```
setsockopt(icmp_sock, IPPROTO_IP, IP_OPTIONS, (char *) &cont_opt, sizeof(cont_opt))
```

Quellcode 4.15: `setsockopt`-Aufruf zum Hinzufügen der IP-Option, aus Datei `cipping.c`

Die Modifizierung des Programms beinhaltet einen neuen Kommandozeilenparameter „`-C session_address`“, der es ermöglicht, die vom Kontextrouter erhaltene *Session* und *ID* zu übermitteln. Wobei der *ID*-Parameter hier als `address` angegeben ist. Momentan entspricht der *ID*-Parameter der IP-Adresse des zu nutzenden Dienstanbieters. Aus diesem Grund erwartet das neue `-C`-Argument eine 16 Bit breite Dezimalzahl `session`, gefolgt von einem Unterstrich (_) und anschließend einer IP-Adresse in *Dotted decimal notation*[2] als `address`-Argument.

Mittels der so übergebenen Parameter ist es dem Programm möglich, die in Kapitel 2.3 und [Deb07] gezeigten IP-Optionen hinzuzufügen. Das modifizierte `ping`-Programm befindet sich auf dem beigelegten Datenträger im Verzeichnis `software/cipping` und trägt den Namen `CIPPING`. Alle bisherigen `ping`-Funktionen und Kommandozeilenparameter bleiben unverändert. Senden und Empfangen der *ICMP*-Pakete erfolgt mit Hilfe eines *Raw Sockets*, wobei durch einen `setsockopt`-Aufruf die übergebenen *Session*- und *ID*-Parameter als IP-Option eingefügt werden. Quellcode 4.15 zeigt den konkreten Aufruf. Wie zu erkennen ist, enthält die Variable `cont_opt` die zu setzende IP-Option.

Das Programm `cont_client` realisiert, wie bereits beschrieben, die Funktionen des in Abbildung 4.3 dargestellten *Kontext-Anwendungsschnittstelle*-Elementes. Das Programm befindet sich im Verzeichnis `software/cont_client` des beiliegenden Datenträgers. Weiterhin basiert der Quellcode des Programms auf den in [Wol06] beschriebenen Beispielquelltexten zur Netzwerk- beziehungsweise Socketprogrammierung aus Kapitel zehn.

Nach dem Start des Programms erzeugt es eine TCP-Socketverbindung zum Routing-Daemon AODV-UU. Im Anschluss ist durch den Nutzer eine Nachricht einzugeben, welche umgehend über den TCP-Socket versandt wird. Gültige Schlüsselwörter sowie deren Argumente sind Anhang A.3 zu entnehmen.

Zu beachten ist weiterhin, dass alle abgesetzten Nachrichten von AODV-UU bestätigt werden. `cont_client` geht daneben von einem abwechselnden Senden und Empfangen von Nachrichten über den aufgebauten TCP-Socket aus. Einige Situation erfordern das Empfangen von mehreren Nachrichten nacheinander von der *Kontext-*

[2] Dezimalschreibweise einer IP Adresse mit Punkt, zum Beispiel 10.0.0.22

Anwendungsschnittstelle. So erzeugt zum Beispiel eine abgesetzte Dienstanfrage mindestens zwei versandte Nachrichten des Routing-Daemons. Direkt nach Eintreffen der Dienstanfrage wird eine Bestätigung übermittelt. Kurz darauf, nach dem Abschließen der Dienstanfrage durch den Routing-Daemon, wird deren Ergebnis ebenfalls versandt.

AODV-UU teilt der *Kontext-Anwendungsschnittstelle* durch das Schlüsselwort „`ACK ++`“ mit, dass eine weitere Nachricht in Kürze übertragen wird. So wartet `cont_client` beim Empfang dieses Schlüsselwortes auf weitere Nachrichten des Routing-Daemons und nimmt während dessen keine Nutzereingaben entgegen.

Enthält eine empfangene Antwortnachricht das Schlüsselwort „`Service-reply from`“, ist von einer erfolgreichen Dienstanfrage auszugehen. In diesem Fall übergibt `cont_client` die gesamte Nachricht der Funktion `create_cont_app_connection`. Diese ist Bestandteil des `cont_client`-Programms und extrahiert aus der übergebenen Nachricht die Adresse des Kontextrouters sowie *ID*- und *Session*-Parameter, welche zum Erstellen der IP-Optionen notwendig sind. Diese Parameter werden nun einer weiteren Funktion `call_cipping` übergeben. Durch `fork`- und `execl`-Systemaufrufe wird das bereits erwähnte `CIPPING`-Programm gestartet.

Alternativ ist die Funktion `create_cont_app_connection` auch vorbereitet, einen gewöhnlichen TCP-Socket zu erstellen und mit Hilfe des `setsockopt`-Befehls die beschriebenen IP-Optionen zu setzen. Dieser Teil des Quellcodes ist momentan auskommentiert. Das Hinzufügen der IP-Optionen im TCP-Socket konnte bereits durch eine Analyse der versandten Pakete im Protokollanalysator *Wireshark* [14] nachgewiesen werden.

In Absprache mit dem Betreuer wurde sich für die `CIPPING`-Variante entschieden, da hier bereits ein Serverprogramm existiert. Marco Wenzel erstellte im Rahmen seiner Diplomarbeit [Wen07b] ein `CIPPING`-Antwortgenerator auf Basis des *click modular router*-Projektes [1].

Ein separates Serverprogramm ist insofern notwendig, da die enthaltenen IP-Optionen bisher unbekannt sind und somit von gewöhnlichen Serveranwendungen unberücksichtigt bleiben. Somit werden die Antwortpakete ohne die beschriebene IP-Option versandt. Für den Kontextrouter wäre es anschließend nicht möglich, die Pakete im Rahmen der Weiterleitungsfunktion korrekt zuzustellen.

Empfängt das `cont_client`-Programm das Schlüsselwort `quitting` in einer Nachricht des Routing-Daemons, beendet es sich selbstständig und schließt die Socketverbindung im Voraus.

4.3.3 Alternatives Konzept einer transparenten Schnittstelle zur Nutzung kontextsensitiver Dienste

Das in Kapitel 4.3 vorgestellte und realisierte Konzept einer Schnittstelle zur Suche nach kontextbehafteten Diensten hat den gravieren Nachteil, dass die verwendeten Anwendungen Kenntnisse über die zu Grunde liegenden Kommunikationsabläufe besitzen müssen. Da sie selbstständig den Dienstanbieter beziehungsweise den Kontextrouter kontaktieren müssen, ist eine Schnittstelle zur Übergabe der Routeradresse sowie den *ID*- und *Session*-Parametern notwendig.

Basierend auf dieser Architektur müsste jede Anwendung, die kontextbehaftete Dienste in Anspruch nehmen will, entsprechend modifiziert werden. Um einen solchen Aufwand zu umgehen, wurde ein Konzept zur transparenten Nutzung von kontextsensitiven Diensten entwickelt.

Abbildung 4.4 visualisiert das hier beschriebene Konzept. Somit ist ein Vergleich, mit dem in Abbildung 4.3 dargestellten realisierten Konzept, leicht möglich.

Wie zu erkennen ist, kommuniziert die lokale Anwendung lediglich mit der *Kontext-Anwendungsschnittstelle*. Hierbei kommt ein virtuelles Netzwerkinterface zum Einsatz, welches durch den *TUN/TAP*-Treiber im Linuxkernel bereit gestellt wird. Im Gegensatz zu gewöhnlichen Netzwerktreibern, die mit der realen Hardware kommunizieren, leitet der *TUN/TAP*-Treiber empfangene Pakete an ein Programm im Userspace weiter und umgekehrt.

Der *TUN/TAP*-Treiber stellt zwei verschiedene virtuelle Netzwerkgeräte zur Verfügung. Wobei TUN ein *Punkt-zu-Punkt*-Netzwerkgerät simuliert, über das Pakete der Schicht drei des OSI-Referenzmodells transportiert werden. Somit ist ein *TUN*-Interface zum Beispiel in der Lage „rohe“ IP-Pakete zu transportieren. Während ein *TAP*-Netzwerkgerät hingegen ein *Ethernet*-Gerät simuliert, welches Pakete der Schicht zwei des OSI-Referenzmodells überträgt.

Zahlreiche OpenSource-Projekte, wie [11], [13] oder [10], nutzen die beschriebenen virtuellen Netzwerkgeräte, um zum Beispiel ein VPN nutzen zu können.

Die in Abbildung 4.4 gezeigte Anwendung kommuniziert lediglich indirekt über ein *TUN*-Netzwerkinterface mit dem Dienstanbieter oder Kontextrouter. Hierbei ist für jede nutzende Anwendung ein *TUN*-Interface zu erzeugen und eine beliebige IP-Adresse zuzuweisen. Somit ist die *Kontext-Anwendungsschnittstelle* in der Lage, mehrere aktive Anwendungen voneinander zweifelsfrei zu unterscheiden. Die abgebildeten roten Pfeile stellen diesen Teil der Kommunikation dar.

Die *Kontext-Anwendungsschnittstelle* nutzt mehrere *Raw Sockets*, um die von den

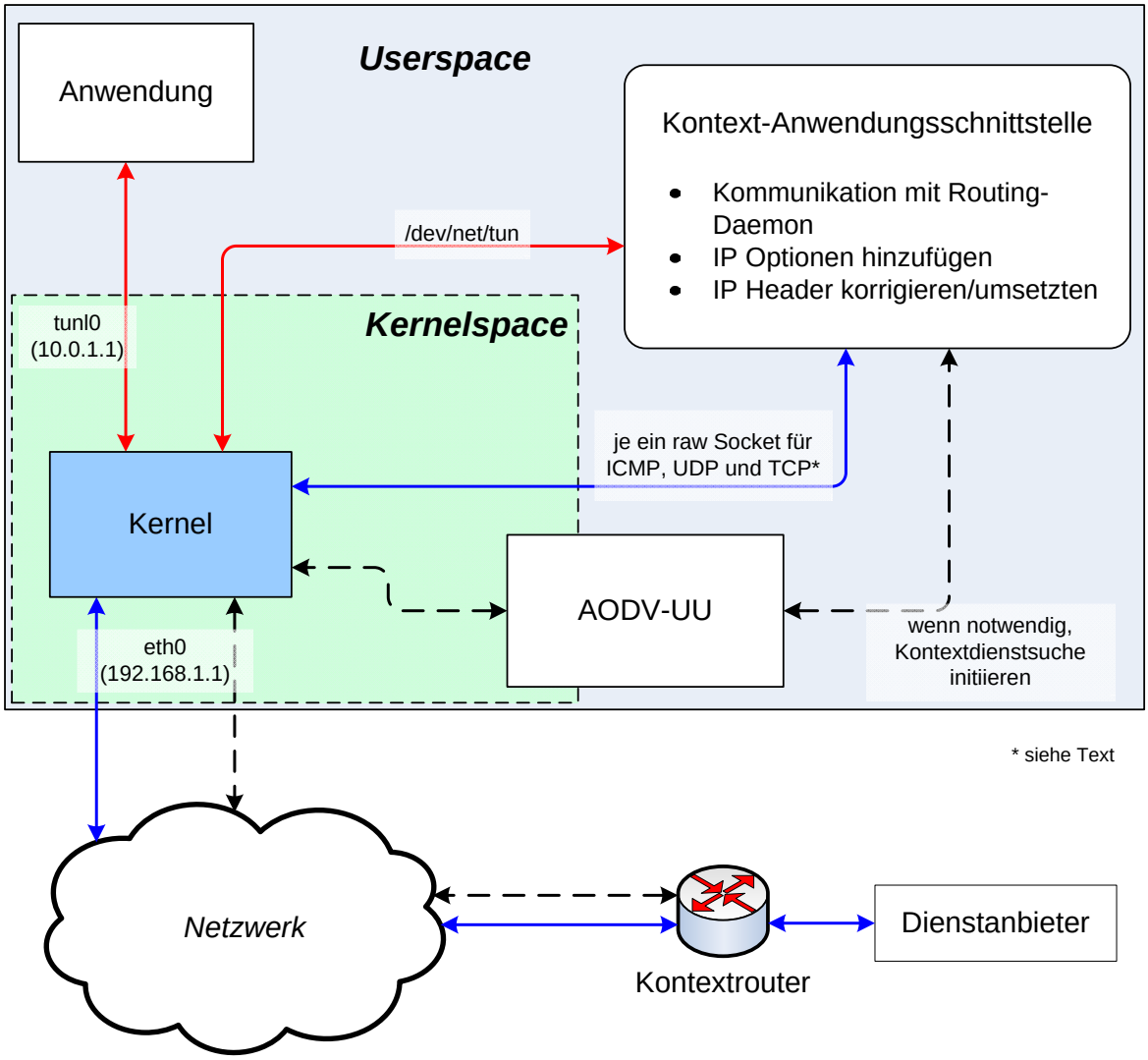

Abbildung 4.4: Schema eines transparenten API zur Nutzung kontextsensitiver Dienste

Anwendungen erhaltenen IP-Pakete an den Kontextrouter oder den Dienstanbieter umzuleiten. Hierzu sind die IP-Header aller eintreffenden Pakete umzusetzen und gegebenenfalls IP-Optionen hinzuzufügen. Die zur Umsetzung notwendigen Informationen, wie Kontextrouteradresse, Session- und ID-Parameter, sind von der Kontextanwendungsschnittstelle im Voraus durch die Initiierung einer Dienstsuche in Erfahrung zu bringen. Die schwarz gestrichelten Pfeile in Abbildung 4.4 symbolisieren diese Anfragen.

Nach erfolgreicher Umsetzung und Korrektur der IP-Header kann die *Kontext-Anwendungsschnittstelle* alle von den Anwendungen erhaltenen IP-Pakete über die erwähnten *Raw Sockets* an den Kontextrouter oder Dienstanbieter versenden. Die abgebildeten blauen Pfeile repräsentieren diesen Kommunikationsteil. Der Einsatz von

Raw Sockets ist damit begründet, dass sie größtmögliche Flexibilität beim Zugriff auf den kompletten Protokollstapel gewährleisten. Jedoch ist hierfür ein etwas höherer Aufwand während der Programmentwicklung zu betreiben, da das Programm die empfangenen IP-Pakete analysieren muss.

Die Nutzung von mehreren *Raw Sockets* ist notwendig, um die Anwortpakete der vom Kontextrouter oder Dienstanbieter empfangen zu können. So ist es laut „*Linux Programmer's Manual*, Sektion 7“[3] das Empfangen von Paketen über *Raw Sockets* nur möglich, wenn beim Erstellen des Sockets eine Protokollnummer angegeben wurde. Um nun das Empfangen nahezu aller praxisrelevanten Protokolle zu ermöglichen, wird je ein *Raw Socket* für die Protokolle *UDP*, *TCP* und *ICMP* erstellt. Falls mehrere Anwendungen gleichzeitig über einen Kontextrouter Dienste nutzen, ist eine Differenzierung der Antwortpakete anhand der enthaltenen IP-Optionen durchzuführen.

Das vorgestellte Konzept spezifiziert nicht die Art und Weise, wie eine Suche kontextbehafteter Dienste initiiert wird. Hierzu sind zwei unterschiedliche Ansätze vorstellbar. Zum Einen kann mittels manueller Eingaben, vergleichbar mit dem implementierten Schnittstellenkonzept aus Abbildung 4.3, eine Dienstsuche erfolgen. Zum Anderen ist es ebenfalls möglich, über die von den Anwendungen genutzte virtuelle Netzwerkschnittstelle (roter Pfeil) Nachrichten an den Routing-Daemon über die *Kontext-Anwendungsschnittstelle* abzusetzen.

Im zweiten Fall benötigen die verwendeten Anwendungen wieder gesonderte Kenntnisse um die *Kontext-Anwendungsschnittstelle* zu kontaktieren und ihr den gesuchten Dienst mit den gewünschten Kontextnummern zu übermitteln. Das Ziel, möglichst vielen Anwendungen die Nutzung der Kontextschnittstelle ohne größeren Aufwand zu ermöglichen, ist mit diesem Ansatz nicht erfüllt.

Als Kompromiss aus beiden Ansätzen wäre eine *Helper-Anwendung* denkbar, die vom Nutzer die gewünschten Dienst- und Kontexttypen abfragt und anschließend über die abgebildete *Kontext-Anwendungsschnittstelle* eine Dienstsuche initiiert. Hierbei kann bereits der Kommunikationsweg über das *TUN*-Interface erfolgen. Nach einer positiven Antwort könnte die eigentliche Anwendung gestartet werden.

Generell muss die verwendete Anwendung über das bereitgestellte virtuelle *TUN*-Interface mit dem Dienstanbieter kommunizieren, da sonst keine transparente Anpassung der IP-Header erfolgen kann.

In Absprache mit dem Betreuer wurde auf eine Implementierung der hier vorgestellten Schnittstelle verzichtet. Eine Überprüfung der Funktionalität der AODV Kon-

[3] Aufruf mittels: `man 7 raw`

texterweiterungen kann ebenfalls durch die realisierten Programme und Schnittstellen erfolgen.

4.4 Hinweise

Wie bereits erwähnt, sind zum Testen der in Abbildung 4.3 gezeigten Kontextanwendung insgesamt drei realisierte Programme notwendig. Als Grundlage kann die besprochene modifizierte AODV-UU-Implementierung angesehen werden. Sie stellt alle Funktionen zum Suchen eines kontextbehafteten Dienstes, durch die in Kapitel 4.3 und 4.3.1 beschriebene Schnittstelle, bereit.

Daneben kommuniziert das Programm `cont_client` über die geschaffene Schnittstelle mit dem Routing-Daemon AODV-UU. Hierzu sind Nutzereingaben notwendig. Im Falle einer erfolgreichen Dienstsuche wird die eigentliche Kontextanwendung `CIPPING` aufgerufen. Alle drei Programme befinden sich im Verzeichnis `software` des beigelegten Datenträgers.

Zum Übersetzen beziehungsweise Kompilieren des Quellcodes sind neben der `gcc`-Compilersuite [5] die dazugehörige *GNU C Library*[4] [4] notwendig. Daneben sind für das erfolgreiche Übersetzten von AODV-UU die Header des verwendeten Linux-Kernels erforderlich.

AODV-UU, sowie dessen Modifizierung, kann durch einen `make`-Aufruf übersetzt werden. Ein anschließender `make install`-Befehl installiert das Programm. Hierbei wird ein Kernelmodul `kaodv` erstellt, welches zum Programmstart geladen und bei Beendigung entladen wird. Zur Ausführen des Programms sind `root`-Privilegien notwendig.

Weiterhin wurde ein neuer Kommandozeilenparameter `-C` hinzugefügt, der ein *CRREQ*-Paket mit einer 58 Bytes langen Erweiterung versendet. Abbildung 2.5 sieht jedoch eine Länge von 56 Bytes vor. Wobei die zusätzlichen zwei Bytes an das Ende der AODV-Paketerweiterung angehangen werden und den Wert „`0`“ enthalten. Diese Änderung am Paketformat ist laut [Wen07b] auf ein softwaretechnisches Problem der Click-Implementierung des Kontextrouters von Marco Wenzel zurückzuführen.

Das Programm `CIPPING` lässt sich ebenfalls mittels eines `make`-Aufrufes im Programmverzeichnis kompilieren. Anschließend ist die erzeugte Binärdatei von Hand in das Verzeichnis des `cont_client`-Programms zu kopieren. Auch hier sind `root`-Pri-

[4] kurz: *glibc*

vilegien zum Ausführen des Programms notwendig. Andernfalls schlägt das Setzen der IP-Optionen mittels `setsockopt` (siehe Quellcode 4.15) fehl. Der neu hinzugefügte Kommandozeilenparameter `-C` schließt eine Nutzung der von `ping` übernommenen Parametern `-R` (*record route*) oder `-T` (*timestamp option*) gleichzeitig aus. Diese beiden Funktionen machen ebenfalls von IP-Optionen Gebrauch und nutzen bereits 39 der maximal 40 zur Verfügung stehenden Bytes für IP-Optionen aus. Somit kann nur einer der Parameter `R`, `T` oder `C` genutzt werden.

Das Programm `cont_client` besteht aus nur einer Quellcodedatei (`cont_client.c`). Zum Übersetzen wurde ein *Shellscript* mit den Namen `make_cont_client.sh` erstellt, welches den notwendigen `gcc`-Aufruf ausführt. Zu beachten ist, dass die `CIPPING`-Binärdatei im gleichen Ordner erwartet wird. Ansonsten schlägt der Aufruf zur `cont_client`-Laufzeit fehl. Alternativ kann auch der Programmpfad in der Funktion `call_cipping` angepasst werden.

Weiterhin wurde eine spezielle *Festnetzversion* der modifizierten AODV-UU Implementierung erstellt. Diese ist für den Einsatz in gewöhnlichen IP-Netzen ohne ad-hoc-Routing konzipiert. Das Programm befindet sich im Verzeichnis `software/aodv-uu-context-festnetz.0.9.5` des beigelegten Datenträgers. In dieser Festnetzversion sind das Senden und Empfangen von gewöhnlichen *AODV RREQ*- und *RREP*-Nachrichten deaktiviert. Daneben kommt das von AODV-UU genutzte Kernelmodul nicht zum Einsatz. Jedoch ist das Senden und Empfangen von erweiterten *RREQ*- sowie *RREP*-Nachrichten über die in Kapitel 4.3 vorgestellte Schnittstelle möglich.

Somit ist gewährleistet, dass auch in gewöhnlichen IP-Netzwerken die beschriebene Infrastruktur der Kontextdienstsuche sowie -nutzung verwendet werden kann. Aufgrund eines Fehlers im Kontextrouters war es bis zur Fertigstellung dieser Arbeit nicht möglich, Antworten auf Dienstanfragen zu empfangen. Durch Analysieren der übertragenen Netzwerkpakete mittels des Protokollanalysators *Wireshark* konnte nachgewiesen werden, dass die versandten *CRREQ*-Pakete einen ordnungsgemäßen Aufbau besitzen und am Kontextrouter eintreffen. Allerdings bleibt das Versenden von *CRREP*-Antwortpaketen aus.

5 Verifikation

Dieses Kapitel beschreibt die Verifikation der implementierten Softwarekomponenten. Hierzu wurden Testszenarien konzipiert, die ausgewählte funktionale Aspekte beleuchten. Es erfolgt eine Beschreibung der Demonstratorumgebung sowie eine Aufschlüsselung der verwendeten Hard- und Softwarebestandteile. Anschließend folgt eine Erläuterung der durchgeführten Messungen und letztendlich die Zusammenfassung und Bewertung der realisierten Szenarien.

5.1 Vorstellung der Testumgebung

Wie in den vorherigen Kapiteln bereits erwähnt, wurde die zur Verifikation der entwickelten Programme verwendete Demonstrator- und Testumgebung im Rahmen der Diplomarbeit [Wen07b] von Marco Wenzel konzipiert und umgesetzt. Wobei vor allem der darin entwickelte *Kontextrouter* zum Einsatz kommt, um die im Rahmen dieser Arbeit erstellten Softwarekomponenten zu verifizieren. Daneben stand ein *click*-Script zur Verfügung, welches einen simplen Kontextdienstanbieter realisiert.

Dieses Script beantwortet *ICMP echo request*-Pakete, welche die in Kapitel 2.3 erwähnten IP-Optionen enthalten. Wobei den Antwortpaketen ebenfalls die IP-Option angefügt wird. Nur so ist es dem Kontextrouter möglich, Pakete einer Kontextanwendungssitzung in beide Richtungen korrekt weiter zu leiten. Den Antwortpaketen eines gewöhnlichen Dienstanbieters, der die genannten IP-Optionen nicht kennt, würden die zusätzlichen IP-Optionen fehlen. Somit käme keine Kommunikation zwischen Client und Server über den Router zustande. Diese realisierte Anwendung wird im Folgenden als *CIPPING* bezeichnet.

Abbildung 5.1 zeigt den Aufbau der verwendeten Demonstratorumgebung. Wie zu erkennen ist, gliedert sich die Testumgebung in drei verschiedene Subnetze. Das links abgebildete *AODV-Netz* besteht zum einen aus den Rechnern *Knoten 1*, *Knoten 2* und *Kontextrouter*. Diese drei Computer spannen ein `10.0.0.0/24`-Subnetz auf, wobei WLAN im *ad-hoc*-Modus als physikalischer Träger zu Einsatz kommt. Zum anderen bilden die Rechner *Knoten 3* und *Kontextrouter* ein zweites Subnetz (`192.168.2.0/24`)

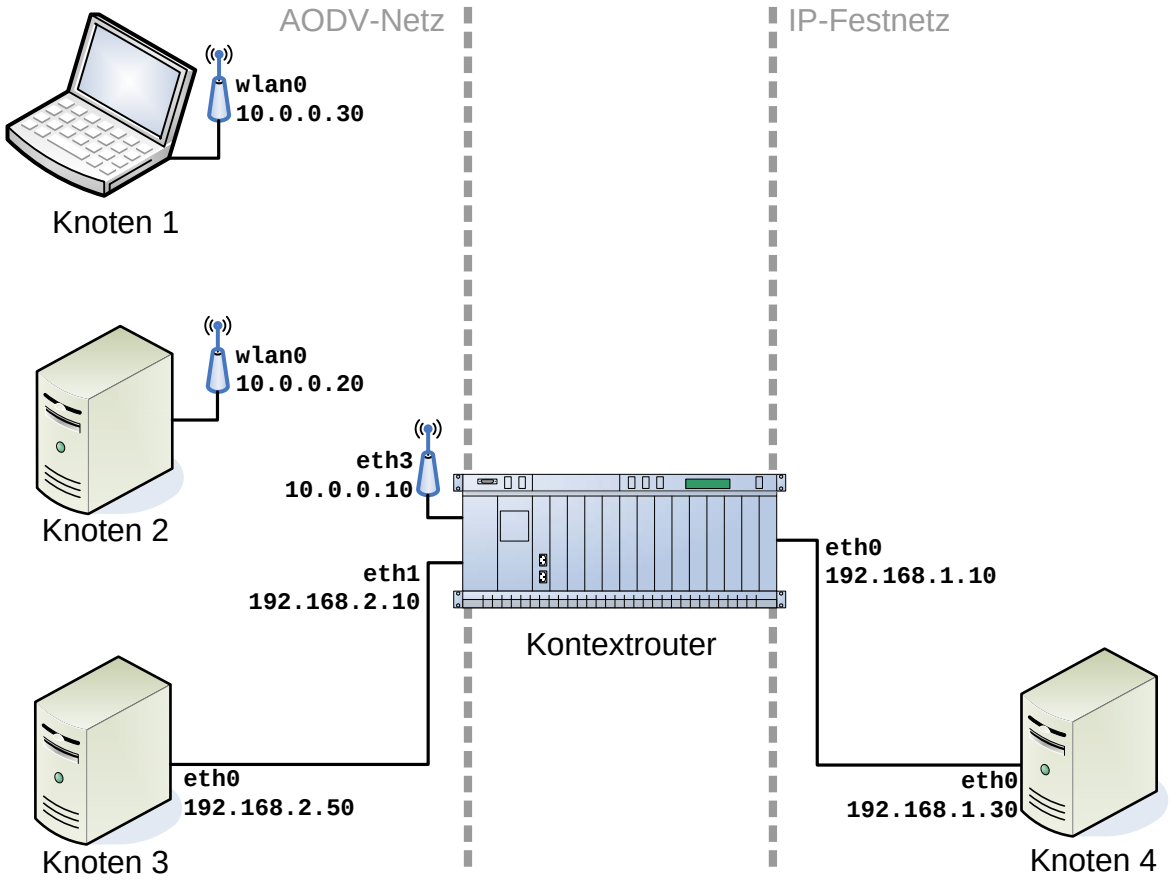

Abbildung 5.1: Aufbau der Demonstratorumgebung

auf *Ethernet*-Basis.

Daneben existiert ein weiteres Subnetz (`192.168.1.0/24`), in dem sich die abgebildeten PCs *Knoten 4* und *Kontextrouter* befinden. Auch hier kommt *Ethernet* als physikalischer Träger zum Einsatz. Abbildung 5.1 zeigt weiterhin die von den Knoten verwendeten Netzwerkgeräte sowie deren zugewiesenen IP-Adressen.

Die auf dem Kontextrouter befindlichen click-Elemente unterscheiden sich von den in [Wen07b] beschriebenen Elementen insofern, dass sie nachträglich von Marco Wenzel erweitert und modifiziert wurden. Diese Änderungen ermöglichen unter anderem die Nutzung von mehreren Netzwerkgeräten mittels des click-Scriptes. Somit können mehrere AODV-Subnetze erstellt werden, die die beschriebenen Kontexterweiterungen unterstützen. Wobei der Kontextrouter das Vermitteln zwischen den einzelnen Netzsegmenten übernimmt.

Weiterhin wurden die click-Basiselemente mit Hilfe des Open-Source-Versionskontrollsystem *GIT* [3] aktualisiert.

Verwendete Hard- und Software

Bei allen verwendeten Knoten handelt es sich um PCs mit der Linuxdistribution *Gentoo* [2] als Betriebssystem. Zur Analyse des Datenverkehrs kommt der Paketanalysator *Wireshark* [14] in Version 0.99.7 zum Einsatz, dessen Ausgaben im Folgenden als Nachweis über gesendete und empfangene Pakete dient. Die Tabellen 5.1 bis 5.5 fassen die technischen Eckdaten der eingesetzten Computer zusammen.

Bezeichnung	**Knoten 1**
CPU, Takt, Cache	Intel Mobile Pentium 4, 2,2GHz, 512kB
Arbeitsspeicher, Swap	756MB, 1537MB
Netzwerkkarten	Intersil Corporation Prism 2.5 (rev 01) (IEEE 802.11b)
Betriebssystem	Gentoo Linux 2.6.23-gentoo-r3 (Gentoo Kernel)
GCC-Version	Gentoo 4.1.2 p1.0.2
AODV-Daemon	AODV-UU 0.9.5 mit Kontexterweiterungen
Netzwerkkonfiguration	wlan0: `[00:E0:00:D1:D0:34]` 10.0.0.30/24 (Ad-hoc-Netz, WLAN)
Beschreibung	Fujitsu Siemens Lifebook E7010, Kontextdienstnehmer im AODV Ad-hoc-Netz

Tabelle 5.1: Hard- und Software – Knoten 1

Knoten 1 übernimmt in nahezu allen Testszenarien die Rolle des Dienstnehmers. Hierbei kommen die vorgestellte AODV-UU-Implementierung mit Kontexterweiterungen zum Einsatz. Das schließt die erstellten Programme `cont_client` und `CIPPING` mit ein. Da es sich bei diesem Knoten um ein Notebook mit integriertem WLAN-Adapter handelt, wird eine mobile Nutzung enorm erleichtert. So befasst sich *Szenario 1* auf Seite 66 zum Beispiel mit der Verifizierung der AODV-Funktionalitäten, wobei *Knoten 1* aus der Sendereichweite des Kontextrouters bewegt wird.

Bezeichnung	**Knoten 2**
CPU, Takt, Cache	Intel Pentium III, 450MHz, 512kB
Arbeitsspeicher, Swap	185MB, 810MB
Netzwerkkarten	Texas Instruments ACX 111 WLAN (IEEE 802.11b/g)
Betriebssystem	Gentoo Linux 2.6.23-gentoo-r3 (Gentoo Kernel)
GCC-Version	Gentoo 4.1.2 p1.0.2
AODV-Daemon	AODV-UU 0.9.5 (unverändert)
Netzwerkkonfiguration	wlan0: `[00:09:5B:BB:36:08]` 10.0.0.20/24 (Ad-hoc-Netz, WLAN)
Beschreibung	Transferknoten ohne Kontexterweiterungen im AODV ad-hoc-Netz

Tabelle 5.2: Hard- und Software – Knoten 2

Knoten 2 dient in allen Testszenarien als Transferknoten. Hierbei leitet dieser eintreffende Pakete über den angeschlossenen WLAN-Adapter weiter. AODV-UU übernimmt hierbei die Verwaltung und Umsetzung der Pakete. Daneben besitzt dieser Knoten keine Kontexterweiterung. Somit kann der Nachweis einer störungsfreien Integration in

gewöhnliche AODV-Netzwerke erbracht werden. Da die eingesetzte WLAN-Hardware nicht vom Linuxkernel von Haus aus unterstützt wird, kommt ein proprietärer Treiber zum Einsatz. Das *Gentoo*-Paket `net-wireless/acx` in Version `0.3.35_p20070101` erzeugt das notwendige Kernelmodul.

Bezeichnung	**Knoten 3**
CPU, Takt, Cache	Intel Pentium II, 233MHz, 512kB
Arbeitsspeicher, Swap	60MB, 251MB
Netzwerkkarten	AMD 79c970, 10/100Mbit Fast Ethernet
Betriebssystem	Gentoo Linux 2.6.23-gentoo-r8 (Gentoo Kernel)
GCC-Version	Gentoo 4.1.2 p1.0.1
AODV-Daemon	Click mit AODV-Elementen und CIPPING-Antwortmodul
Netzwerkkonfiguration	eth0: `[00:A0:D2:18:A0:59]` 192.168.2.50/24 (Ad-hoc-Netz, Ethernet)
Beschreibung	Dienstanbieter im AODV-Festnetz (`CIPPING`)

Tabelle 5.3: Hard- und Software – Knoten 3

Knoten 3 ist via Ethernet mit dem Kontextrouter verbunden. In diesem Subnetz kommt ebenfalls AODV zum Einsatz. Ein Click-Script übernimmt hier die Aufgabe des AODV-Daemons und stellt daneben den bereits erwähnten *CIPPING*-Dienst zur Verfügung. Zusätzlich ist durch den in Abbildung 5.2 gezeigten Befehl die Beantwortung von *ICMP echo request*-Paketen des Kernels deaktiviert. Ansonsten würde sowohl der Linuxkernel, als auch das Click-Script die eintreffenden *ICMP echo request*-Pakete des `CIPPING`-Programms beantworten. Hierbei enthält ein vom Kernel generiertes Anwortpaket nicht die beschriebenen IP-Optionen.

```
echo "1" > /proc/sys/net/ipv4/icmp_echo_ignore_all
```

Abbildung 5.2: Befehl zum Ignorieren von *ICMP echo request*-Paketen durch den Linuxkernel

Bezeichnung	**Knoten 4**
CPU, Takt, Cache	Intel Celeron, 300MHz, 128kB
Arbeitsspeicher, Swap	186MB, 810MB
Netzwerkkarten	Realtek 8029 Ethernet
Betriebssystem	Gentoo Linux 2.6.23-gentoo-r3 (Gentoo Kernel)
GCC-Version	Gentoo 4.1.2 p1.0.1
AODV-Daemon	ohne
Netzwerkkonfiguration	eth0: `[00:05:5D:D3:D5:EC]` 192.168.1.30/24 (IP-Festnetz)
Beschreibung	Dienstanbieter im IP-Festnetz (`CIPPING`) durch click-Script

Tabelle 5.4: Hard- und Software – Knoten 4

Knoten 4 ist ebenfalls via Ethernet mit dem Kontextrouter verbunden. Es kommt

jedoch kein AODV zum Einsatz. Wie *Knoten 3* stellt auch dieser Rechner den beschriebenen `CIPPING`-Dienst mit Hilfe eines click-Scriptes bereit. Somit ist hier ebenfalls durch den in Abbildung 5.2 gezeigten Befehl die Beantwortung von *ICMP echo request*-Paketen im Kernel deaktiviert.

Daneben bietet dieser Knoten die Möglichkeit zum Einsatz der in Abschnitt 4.4 beschriebene Festnetzvariante der AODV-UU Modifizierung. Testszenario fünf beschäftigt sich mit diesem Anwendungsfall.

Bezeichnung	**Kontextrouter**
CPU, Takt, Cache	Intel Pentium III, 800MHz, 256kB
Arbeitsspeicher, Swap	256MB, 512MB
Netzwerkkarten	3Com 3c905B, 10/100Mbit/s Fast Ethernet (2 mal) Dell TrueMobile 1150 Series PC Card Ver. 01.01 (IEEE 802.11b)
Betriebssystem	Gentoo Linux 2.6.22-gentoo-r9 (Gentoo Kernel)
GCC-Version	Gentoo 4.1.2 p1.0.1
AODV-Daemon	Click GIT vom 20.12.2007 (Userlevel) mit AODV-Elementen und Kontexterweiterungen
Netzwerkkonfiguration	eth0: `[00:50:04:EE:96:52]` 192.168.1.10/24 (IP-Festnetz) eth1: `[00:50:04:EE:95:A8]` 192.168.2.10/24 (Ad-hoc-Netz, Ethernet) eth3: `[00:02:2D:49:8C:FF]` 10.0.0.10/24 (Ad-hoc-Netz, WLAN)
Beschreibung	Kontextrouter

Tabelle 5.5: Hard- und Software – Kontextrouter

Der Kontextrouter stellt das zentrale Element der in Abbildung 5.1 gezeigten Demonstratorumgebung dar. Auf Basis der in [Wen07b] implementierten click-Elemente und Scripte ist der Kontextrouter in der Lage, IP-Pakete zwischen allen gezeigten Subnetzen zu vermitteln. Nach Beendigung von [Wen07b] modifizierte Marco Wenzel zahlreiche click-Elemente. Diese Änderungen hatten unter anderem das Ziel, den AODV-Betrieb an mehrere Netzwerkschnittstellen zu ermöglichen. Wie in Kapitel 4.4 bereits genannt, ergeben sich hieraus einige Einschränkungen des Funktionsumfangs im Vergleich zu [Wen07b]. Daneben wurden die click-Basiselemente mit Hilfe des Open-Source-Versionskontrollsystems *GIT* zuletzt am 20.12.2007 aktualisiert.

Da click auf dem Kontextrouter ein eigenes *ICMP echo request*-Antwortmodul bereit stellt, ist wie auf *Knoten 3* und *Knoten 4* die Beantwortung dieser Pakete durch den Kernel deaktiviert. Abbildung 5.2 zeigt den hierfür notwendigen Befehl. Daneben kommt das click-Script `context_router_3if_20080207.click` in allen Testszenarien zum Einsatz.

[Wen07b] nutzt eine Implementierung des AODV-Protokolls im Userspace. Momentan erfährt der Userlevel-AODV-Daemon nicht, wenn durch ein lokales Programm eine neue Route zu einem Host im AODV-Netz gesucht werden soll. Für eintreffenden

Datenverkehr benachbarter Knoten ist das nicht relevant, da click die Pakete entgegennimmt und gegebenenfalls eine Routensuche initiiert. Somit ist der Kontextrouter nicht als Dienstanbieter geeignet. Diese Einschränkung beeinflusst die folgenden Testszenarios nicht, da der Kontextrouter entweder eintreffende IP-Pakete vermittelt oder sie selbst von click beantwortet werden.

Hinweis: alle zur Ausführung kommenden Client- und Serverprogramme auf sämtlichen gezeigten Knoten besitzen `root`-Privilegien.

5.2 Szenarien zur Funktionsprüfung

Das folgende Kapitel soll der Verifizierung der neu erstellten und modifizierten Programme dienen und die Kommunikationsabläufe veranschaulichen. Die folgenden Testszenarien beleuchten ausgewählte Aspekte der Kommunikation zwischen den im Rahmen dieser Arbeit erstellten Client und dem bereits vorhandenen Kontextrouter und Dienstanbieter.

Zur Überprüfung der in den einzelnen Testszenarien gesetzten Ziele dienen Screenshots des Protokollanalysators *Wireshark* [14] sowie Debuggausgaben der verwendeten Programme. Als genutzte Dienste kommen in nahezu allen Testszenarien `ICMP echo request`- und `ICMP echo reply`-Nachrichten zum Einsatz. Diese Pakete werden vom Programm `ping` erzeugt und entweder direkt vom Betriebssystemkern oder von click beantwortet. `ping` dient in erster Linie zur Netzwerkanalyse und Fehlersuche. Es liefert unter anderem Informationen zu ICMP-Fehlermeldungen, Paketverlustquoten, Sequenznummern, Informationen über Paketgrößen/Fragmentierung, Time-to-live und Round-Trip-Time. Die hier verwendete `ping`-Variante stammt aus dem *Gentoo*-Paket `net-misc/iputils` in Version `20070202`. Als Kommandozeilenargumente kommen sowohl bei `ping` als auch dessen Modifizierung `CIPPING` die Parameter `-i2 -n` zum Einsatz. Hierbei wird durch `-i2` das Intervall der versandten *ICMP echo request*-Pakete auf zwei Sekunden erhöht und mittels `-n` sämtliche DNS-Abfragen deaktiviert.

5.2.1 Szenario 1: AODV Funktionalität

Dieses Testszenario soll die gewöhnliche AODV-Funktionalität der modifizierten AODV-UU-Implementierung nachweisen. Hierzu werden vom *Knoten 1 ICMP echo request*-Pakete an den Kontextrouter mittels folgendem Kommando versandt: `ping -i2 -n -R 10.0.0.10`. Der Parameter `-R` veranlasst `ping` die IP-Option *record rou-*

te an die versandten *ICMP*-Pakete anzufügen. Hierdurch ist der Weg des IP-Paketes nachvollziehbar. Zu beachten ist, dass das `ping`-Antwortelement in der aktuellen Konfiguration des Kontextrouters diese IP-Option nicht berücksichtigt. Somit erscheint die IP-Adresse des Kontextrouters nicht in der *ping*-Ausgabe, da sie nicht in der *record route*-IP-Option enthalten ist.

Abbildung 5.3 zeigt den schematischen Ablauf des Testszenarios. Die abgebildeten Kreise symbolisieren die jeweilige Reichweite der WLAN Adapter.

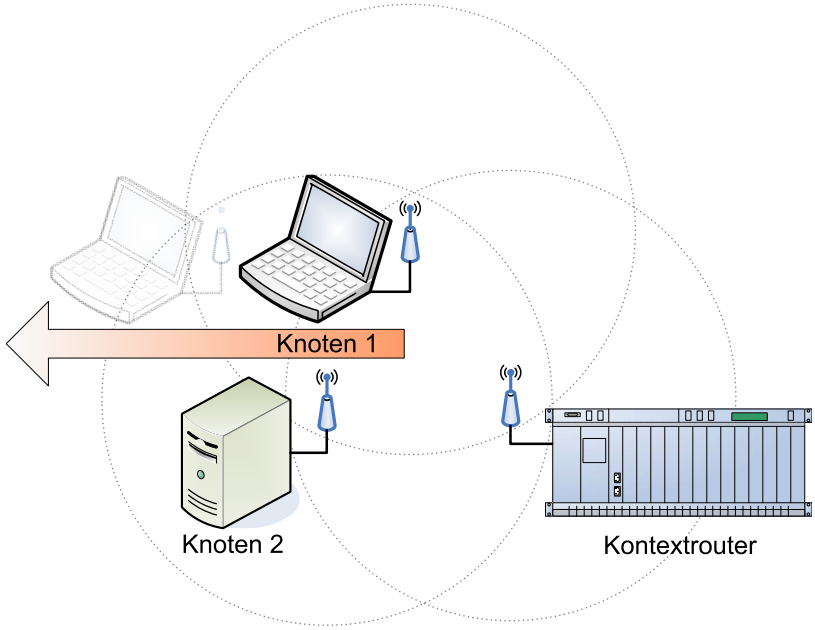

Abbildung 5.3: Aufbau Testszenario 1

Wie zu erkennen ist, befindet sich *Knoten 1* anfangs noch in Funkreichweite des Kontextrouters. Aufgrund der Bewegung von *Knoten 1* ist nach kurzer Zeit keine direkte Kommunikation mit dem Kontextrouter möglich. Dieses Ereignis muss vom AODV-Deamon auf *Knoten 1* erkannt werden, um im Anschluss eine neue Route zum Kontextrouter zu suchen. Während der Routensuche sollten alle an den jeweiligen Knoten zu sendenden Pakete von der modifizierten AODV-UU-Implementierung zwischengespeichert werden. Dieser Vorgang ist im Folgenden als *Handover* bezeichnet.

Abbildung 5.4 zeigt Auszüge der `ping`-Bildschirmausgaben. Wie an den ersten drei abgebildeten `ping`-Meldungen zu erkennen ist, erreicht *Knoten 1* (10.0.0.30) den Kontextrouter (10.0.0.10) auf direktem Weg. Die beschriebene Initiierung der Routensuche, findet nach dem Empfang des dreizehnten *ICMP echo reply*-Paketes (`icmp_seq=13`) statt. So ist die *Round-Trip-Time* des folgenden Paketes um ein Vielfaches höher. Daneben ist der `ping`-Ausgabe zu entnehmen, dass dieses Paket *Knoten 2* (10.0.0.20)

```
PING 10.0.0.10 (10.0.0.10) 56(124) bytes of data.
64 bytes from 10.0.0.10: icmp_seq=1 ttl=255 time=2.96 ms
NOP
RR: 10.0.0.30
    10.0.0.30

64 bytes from 10.0.0.10: icmp_seq=2 ttl=255 time=3.14 ms
NOP (same route)
64 bytes from 10.0.0.10: icmp_seq=3 ttl=255 time=2.96 ms
NOP (same route)

...

64 bytes from 10.0.0.10: icmp_seq=13 ttl=255 time=3.22 ms
NOP (same route)
64 bytes from 10.0.0.10: icmp_seq=14 ttl=254 time=451 ms
NOP
RR: 10.0.0.30
    10.0.0.20
    10.0.0.20
    10.0.0.30

64 bytes from 10.0.0.10: icmp_seq=15 ttl=254 time=5.45 ms
NOP (same route)
64 bytes from 10.0.0.10: icmp_seq=16 ttl=254 time=5.51 ms
NOP (same route)

...

--- 10.0.0.10 ping statistics ---
90 packets transmitted, 82 received, +2 duplicates, +5 errors,
 8% packet loss, time 178001ms
rtt min/avg/max/mdev = 2.936/12.452/451.309/48.570 ms, pipe 2
```

Abbildung 5.4: Auszug der `ping`-Bildschirmausgaben – Testszenario 1

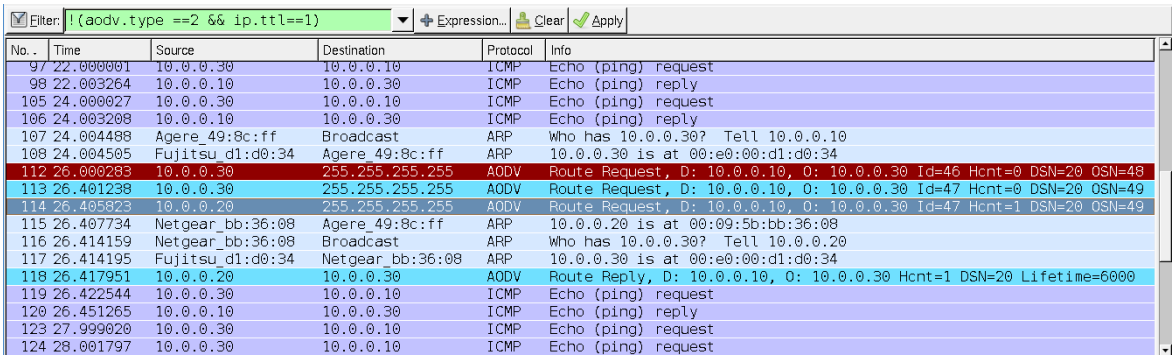

Abbildung 5.5: Wireshark Screenshot – Testszenario 1

passierte. Weiterhin ist der `ttl`-Wert, im Vergleich zu vorherigen Paketen, um eins erniedrigt.

Abbildung 5.5 zeigt die Routensuche nach dem Verlassen der Funkreichweite des Kontextrouters im Protokollanalysator Wireshark. Der eingestellte Filter „`!(aodv.type==2 && ip.ttl==1))`“ blendet die sekündlich versandten *AODV-Hello*-Nachrichten aus. Die abgebildeten Pakete `112` und `113` zeigen die von *Knoten 1* versandten *AODV RREQ*-Nachrichten zur Suche einer neuen Route. Tabelle 5.6 listet die von Wireshark verwendeten Abkürzungen für AODV-Pakete auf. Wie zu erkennen ist, suchen die abgebildeten RREQ-Nachrichten nach dem Kontextrouter (`D: 10.0.0.10`). Dem Paket `114` ist hingegen zu entnehmen, dass *Knoten 2* das empfangene RREQ erneut versendet. Kurze Zeit später trifft mit dem Paket `118` eine RREP-Nachricht mit dem gewünschten Ziel (`D: 10.0.0.10`) von *Knoten 2* (`10.0.0.20`) ein. Somit ist *Knoten 1* eine aktive Route zum Kontextrouter bekannt und die zurückgehaltenen Pakete können versendet werden.

Kürzel	**Beschreibung**
D:	Destination IP
O:	Originator IP
ID:	RREQ ID
Hcnt:	Hop count
DSN:	Destination Sequence Number
OSN:	Originator Sequence Number

Tabelle 5.6: Wireshark-Kürzel für AODV Pakete

Die Informationen aus Abbildungen 5.4 und 5.5 weisen die ordnungsgemäße AODV-Funktionalität der modifizierten AODV-UU-Implementierung nach. Weiterhin zeigt dieses Testszenario das erfolgreiche Zusammenwirken verschiedener AODV-Implementierungen.

Die vollständige `ping`-Bildschirmausgabe sowie das *Wireshark-capture File* sind dem beiliegenden Datenträger aus Verzeichnis `/messungen/Szenario1` zu entnehmen.

5.2.2 Szenario 2: Kontextsensitive Routinganfragen und -antworten in einem heterogenen AODV Netzwerk

Dieses Testszenario dient der Verifizierung von kontextbehafteten Dienstanfragen und -antworten mittels der in Kapitel 2.3 vorgestellten *CRREQ*- und *CRREP*-Nachrichten. Hierbei startet *Knoten 1* eine Dienstanfrage an den Kontextrouter. Die beiden Knoten kommunizieren ausschließlich über *Knoten 2*. Der Kommunikationsablauf ent-

spricht dem Endzustand von Testszenario eins. Der Begriff des „heterogenen AODV Netzwerks“ bezieht sich auf den Einsatz verschiedenster AODV-Implementierungen. Dieses Testszenario erprobt einen Mischbetrieb aus AODV-Knoten mit und ohne Konterweiterung. Abbildung 5.6 zeigt den schematischen Aufbau dieses Testszenarios.

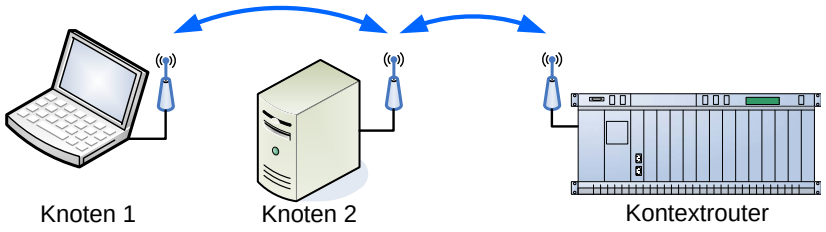

Abbildung 5.6: Aufbau Testszenario 2

Knoten 1, im Folgenden als Client bezeichnet, sucht nach dem Diensttyp „30“ und der in Tabelle 5.7 gezeigten Kontexttypen und Prioritäten. Die Dienstnummer „30“ ist rein willkürlich gewählt und soll im Folgenden lediglich der Funktionsüberprüfung dienen.

Kontexttyp	1	2	3	4	5	6	7	8	9	10
Priorität	7	5	4	6	3	2	2	3	4	0

Tabelle 5.7: Geforderte Kontexttypen und Prioritäten des Clients – Testszenario 2

In Tabelle 5.8 sind alle am Kontextrouter registrierten Dienste dargestellt. Im Vergleich zu Abbildung 5.1 auf Seite 62 ist zu sehen, dass sich einer der registrierten Dienstanbieter im IP-Festnetz befindet und ein anderer im ethernetbasierten AODV-Subnetz. Nach Abschluss der Dienstanfrage wird auf eine Nutzung der angebotenen Dienste verzichtet, da dieses Testszenario ausschließlich der Verifikation von Dienstsuchemechanismen dient.

IP-Adresse	**Diensttyp**	**Kontexttypen**
192.168.2.50	30	4,5,7,8
192.168.1.30	30	1,2,3

Tabelle 5.8: Registierte Dienste – Testszenario 2

Die Dienstsuche wurde mittels der bereits vorgestellten Software `cont_client` initiiert. Der in Abschnitt 4.4 erwähnte neue AODV-UU-Kommandozeilenparameter `-C` kam hier, wie in allen folgenden Testszenarios auch, zum Einsatz.

Abbildung 5.7 zeigt getätigte Eingaben sowie resultierende Ausgaben des Programms `cont_client`. Zur besseren Lesbarkeit sind ursprünglich lange Zeilen umgebrochen.

Dies ist durch das Zeichen „\“ markiert. Wie ersichtlich ist, befindet sich der Dienstanbieter `192.168.1.30` in der lokalen Liste für unerwünschte Server (Blacklist). Dienstangebote dieses Servers werden zurückgewiesen. Hierbei repräsentiert das Schlüsselwort `ID` momentan die IP-Adresse des Servers. In der Antwortnachricht des Routers sind zusätzlich der gesuchte Dienst `Service`, die vergebene `Session` zur Weiterleitung des Anwendungsdatenverkehrs sowie die vom Dienstanbieter unterstützen Kontexttypen enthalten. Das abgebildeten `NA`-Schlüsselwort gibt an, ob das *No Alternatives*-Flag im empfangenen *CRREP*-Paket aktiv war oder nicht. Ist dies der Fall, ist es dem Kontextrouter nicht möglich, weitere alternative Dienstanbieter zu ermitteln.

```
# ./cont_client localhost
Nachricht zum Versenden: blacklist-print
MSG from routing daemon: ACK -- Blacklist entrys:
192.168.1.30

Nachricht zum Versenden: context-request router=10.0.0.10 \
  service=30 ct=1,7:2,5:3,4:4,6:5,3:6,2:7,2:8,3:9,4:10,0
MSG from routing daemon: ACK ++ requesting context service...

MSG from routing daemon: Service-reply from Router=10.0.0.10,\
  Service=30, ID=192.168.2.50, Session=2, CT=4,5,7,8, NA=1

Nachricht zum Versenden: quit
MSG from routing daemon: ACK -- Quitting...
```

Abbildung 5.7: `cont_client`-Bildschirmausgaben – Testszenario 2

In Abbildung 5.8 ist der gesamte Kommunikationsablauf dieser Dienstanfrage zu erkennen. Es ist ersichtlich, dass kurz nach Erhalt der ersten Antwort ein weiteres *RREQ*-Paket versandt wird. Dieses Verhalten deutet auf ein Vorhandensein des ersten erhaltenen Dienstanbieters in der lokalen Serverblackliste hin. Die kurze Zeitspanne zwischen Eintreffen des ersten *RREP*-Paketes und dem Versand des zweiten *RREQ* schließt ein manuelles Versenden durch den Nutzer nahezu aus.

Zu Beginn zeigt jedoch Abbildung 5.9 das erste versandte *CRREQ*-Paket, welches in Abbildung 5.8 die Paketnummer `68` trägt. Diese *CRREQ*-Nachricht wurde bereits von *Knoten 2* empfangen und anschließend weitergeleitet, so dass es den Kontextrouter erreicht.

Der im unteren Teil der Abbildung 5.9 blau markierte Bereich entspricht der in Kapitel 2.3 beschriebenen AODV-Kontextprotokollerweiterung. Abbildung 2.5 beschreibt den konkreten Paketaufbau der Erweiterung.

Wie zu erkennen ist, trägt die Erweiterung den Typ `16` und ist abzüglich Typ- und Längenfeld `58` Bytes groß. Die Länge von `58` anstatt der `56` vorgeschlagenen By-

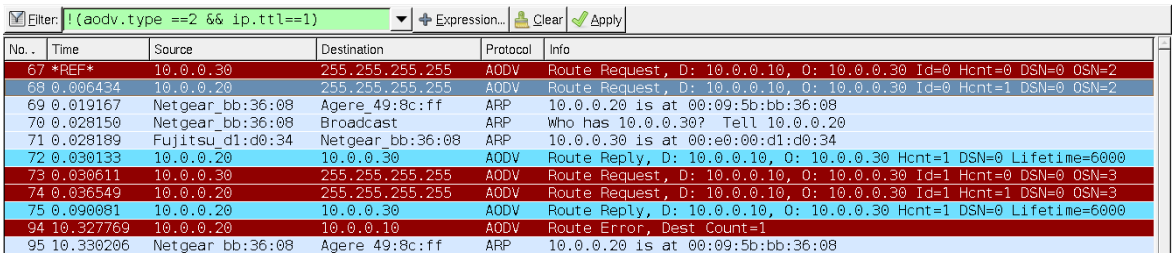

Abbildung 5.8: Wireshark Screenshot – Testszenario 2, Übersicht

tes, ergibt sich aus einem softwaretechnischen Problem der click-Implementierung. Der dargestellte hexadezimale Wert `00 1e`[1] entspricht dem gesuchten Diensttyp „`30`“. Anschließend sind die gewünschten Kontexttypen und deren Prioritäten dargestellt. Die in Tabelle 5.7 gezeigten Kontextnummern und Prioritäten entsprechen dem hexadezimalen Wert `fd ce ba ab c8`, welcher ebenfalls in Abbildung 5.9 zu finden ist.

Abbildung 5.10 zeigt die AODV-Protokollerweiterung des *CRREP*-Paketes im Detail. Abbildung 2.6 beschreibt den konkreten Paketaufbau der Erweiterung. Der Kontextrouter erzeugt *RREP*-Paketerweiterung der Länge `22` statt der vorgeschriebenen `21` Byte. Auch dies ist auf die bereits erwähnten softwaretechnischen Probleme zurückzuführen. Der abgebildete hexadezimale Wert `c0 a8 01 1e` entspricht dem `Identifier`-Feld und somit der IP-Adresse (`192.168.1.30`) des Dienstanbieters in *Network Byte Order*.

Somit ist dieser angebotene Server in der lokalen Blacklist vorhanden. Wie Abbildung 5.8 zu entnehmen ist, sendet AODV-UU sofort nach dem Eintreffen des ersten *CRREP*-Paketes ein weiteres *CRREQ*. Das in Abbildung 5.11 dargestellte Paket ist wie das im Abbildung 5.9 gezeigt *CRREQ* von *Knoten 2* erzeugt wurden. Während des Betrachtens der blau markierten Paketerweiterung fällt auf, dass das `ID`-Feld nach der Längenangabe mit dem aus Abbildung 5.10 übermittelten Wert `c0 a8 01 1e` gefüllt ist. Die Angaben zu gewünschten Kontexttypen sowie deren Prioritäten sind identisch mit dem ersten versandten *CRREQ*-Paket, siehe Abbildung 5.9.

Das im Anschluss empfangene *CRREP*-Paket enthält die in Abbildung 5.7 gezeigten Werte wie `ID`, `Session`, Dienst und unterstützte Kontexttypen. Der gezeigte hexadezimale Wert der vom Dienstanbieter angebotenen Kontextnummern `1b` entspricht ebenfalls der in Tabelle 5.8 und Abbildung 5.7 genannten Kontexttypen.

In Abbildung 5.7 ist weiterhin der Ausdruck `NA=1` in der Rückmeldung des AODV-Daemons zu erkennen. Dies weist wie bereits erwähnt darauf hin, dass kein weiterer Server verfügbar ist. Das so genannte „*No Alternatives*-Flag“ ist wie Kapitel 2.3 zeigt

[1] *Network Byte Order*

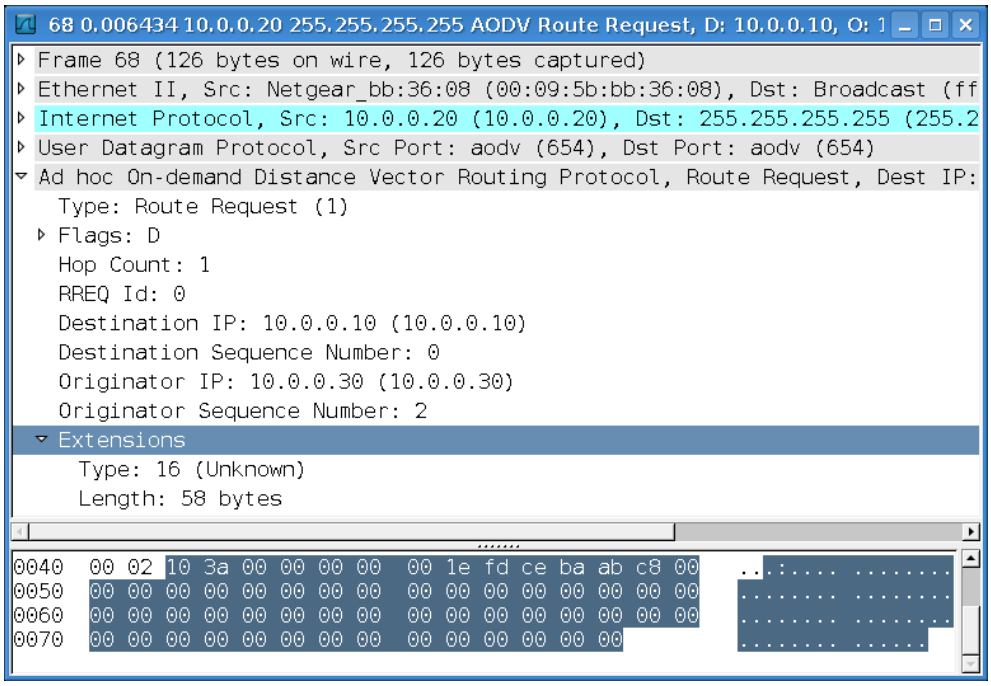

Abbildung 5.9: Wireshark Screenshot – Testszenario 2, CRREQ1

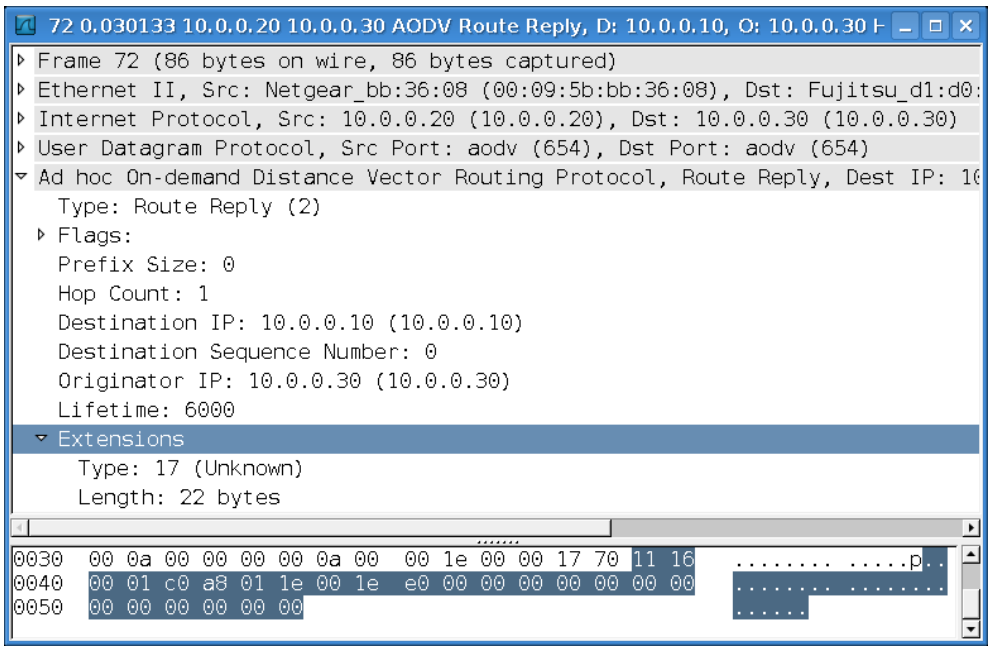

Abbildung 5.10: Wireshark Screenshot – Testszenario 2, CRREP1

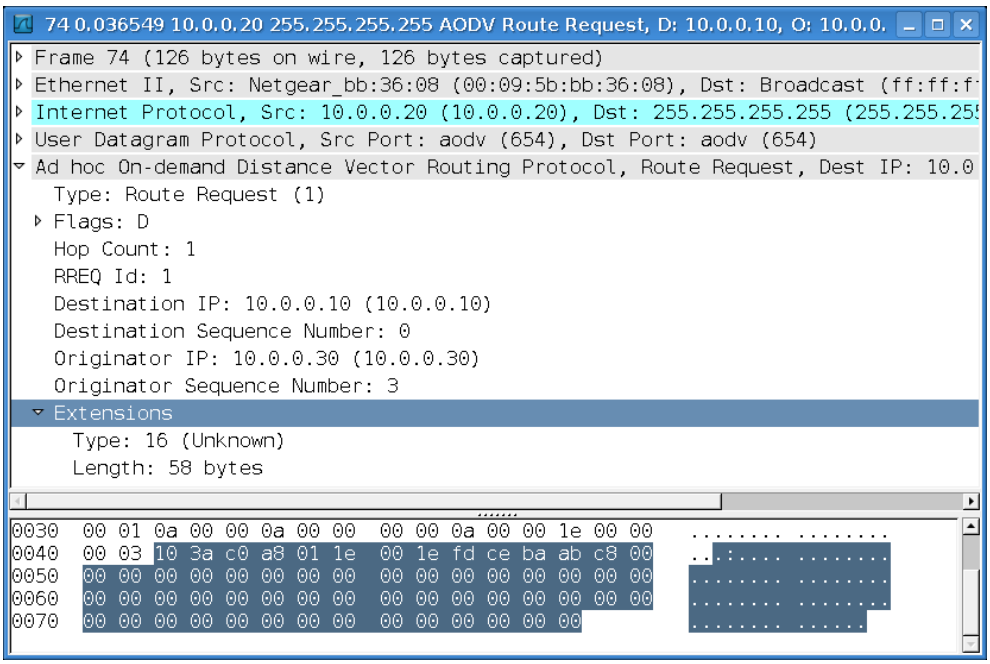

Abbildung 5.11: Wireshark Screenshot – Testszenario 2, CRREQ2

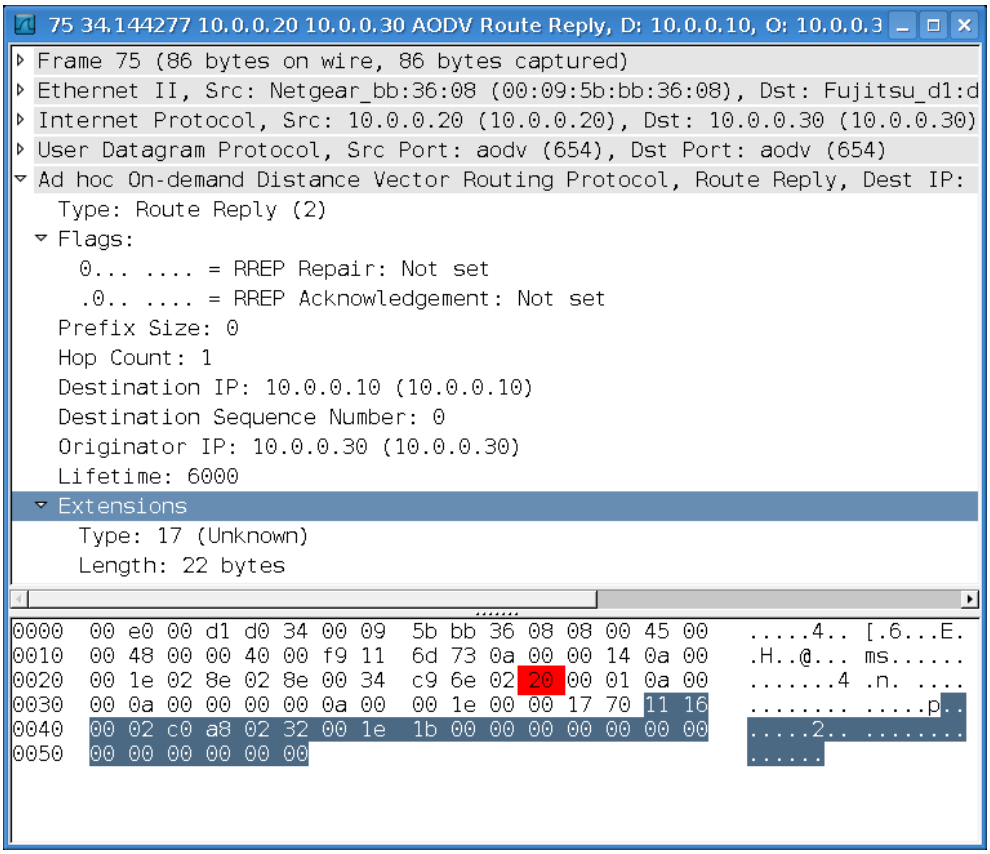

Abbildung 5.12: Wireshark Screenshot – Testszenario 2, CRREP2

```
18:56:01.660 rreq_create: Assembled RREQ 10.0.0.10
18:56:01.660 log_pkt_fields: rreq->flags:D rreq->hopcount=0 rreq->rreq_id=0
18:56:01.661 log_pkt_fields: rreq->dest_addr:10.0.0.10 rreq->dest_seqno=0
18:56:01.661 log_pkt_fields: rreq->orig_addr:10.0.0.30 rreq->orig_seqno=2
18:56:01.661 aodv_socket_send: AODV msg to 255.255.255.255 ttl=2 size=84
18:56:01.661 c_rreq_route_discovery: Seeking 10.0.0.10 ttl=2 Service=30  \
  CT=1,7:2,5:3,4:4,6:5,3:6,2:7,2:8,3:9,4:10,0
18:56:01.691 aodv_socket_process_packet: Received RREP
18:56:01.691 rrep_process: from 10.0.0.20 about 10.0.0.30->10.0.0.10
18:56:01.691 log_pkt_fields: rrep->flags: rrep->hcnt=1
18:56:01.691 log_pkt_fields: rrep->dest_addr:10.0.0.10 rrep->dest_seqno=0
18:56:01.691 log_pkt_fields: rrep->orig_addr:10.0.0.30 rrep->lifetime=6000
18:56:01.691 rrep_process: RREP include CONTEXT EXTENSION

18:56:01.691 rreq_create: Assembled RREQ 10.0.0.10
18:56:01.691 log_pkt_fields: rreq->flags:D rreq->hopcount=0 rreq->rreq_id=1
18:56:01.691 log_pkt_fields: rreq->dest_addr:10.0.0.10 rreq->dest_seqno=0
18:56:01.691 log_pkt_fields: rreq->orig_addr:10.0.0.30 rreq->orig_seqno=3
18:56:01.692 aodv_socket_send: AODV msg to 255.255.255.255 ttl=2 size=84
18:56:01.692 c_rreq_route_discovery: Seeking 10.0.0.10 ttl=2 Service=30  \
  CT=1,7:2,5:3,4:4,6:5,3:6,2:7,2:8,3:9,4:10,0
18:56:01.692 rt_table_insert: Inserting 10.0.0.10 (bucket 10) next hop 10.0.0.20
18:56:01.692 nl_send_add_route_msg: ADD/UPDATE: 10.0.0.10:10.0.0.20 ifindex=3
18:56:01.693 rt_table_insert: New timer for 10.0.0.10, life=6000
18:56:01.751 aodv_socket_process_packet: Received RREP
18:56:01.751 rrep_process: from 10.0.0.20 about 10.0.0.30->10.0.0.10
18:56:01.751 log_pkt_fields: rrep->flags: rrep->hcnt=1
18:56:01.751 log_pkt_fields: rrep->dest_addr:10.0.0.10 rrep->dest_seqno=0
18:56:01.751 log_pkt_fields: rrep->orig_addr:10.0.0.30 rrep->lifetime=6000
18:56:01.751 rrep_process: RREP include CONTEXT EXTENSION
```

Abbildung 5.13: Auszug des AODV-UU-Logfile – Testszenario 2

direkt nach den gewöhnlichen Flags des *RREP*-Paketes angesiedelt. Abbildung 5.12 zeigt zusätzlich im rot markierten Bereich das aktivierte *No Alternatives*-Flag.

Abbildung 5.13 stellt weiterhin Ausschnitte des AODV-UU-Logfile dar. Hierbei trennt die eingefügte Leerzeile das Eintreffen des ersten *CRREP*-Paketes und das Versenden des zweiten *CRREQ* voneinander ab.

Die hier präsentierten Wireshark-Screenshots und Programmausgaben belegen das korrekte Erstellen sowie Verarbeiten der in Kapitel 2.3 vorgestellten Protokollerweiterungen. Daneben konnte die neu erstellte Blacklistfunktion und die Erkennung des „*No Alternatives*-Flag“ erfolgreich demonstriert werden.

Das vollständige *Wireshark-capture File*, die `cont_client`-Bildschirmausgabe sowie die komplette AODV-UU-Log Datei sind dem beiliegenden Datenträger aus Verzeichnis `/messungen/Szenario2` zu entnehmen.

5.2.3 Szenario 3: Weiterleitung des kontextsensitiven Datenverkehrs und Rerouting

Dieses Testszenario dient der Veranschaulichung der Dienstnutzung. Hierbei erfolgt eine Dienstanfrage durch *Knoten 1* an den Kontextrouter, wobei der gesamte Datenverkehr über *Knoten 2* abgewickelt wird. Abbildung 5.14 beschreibt den schematischen Aufbau dieses Szenarios.

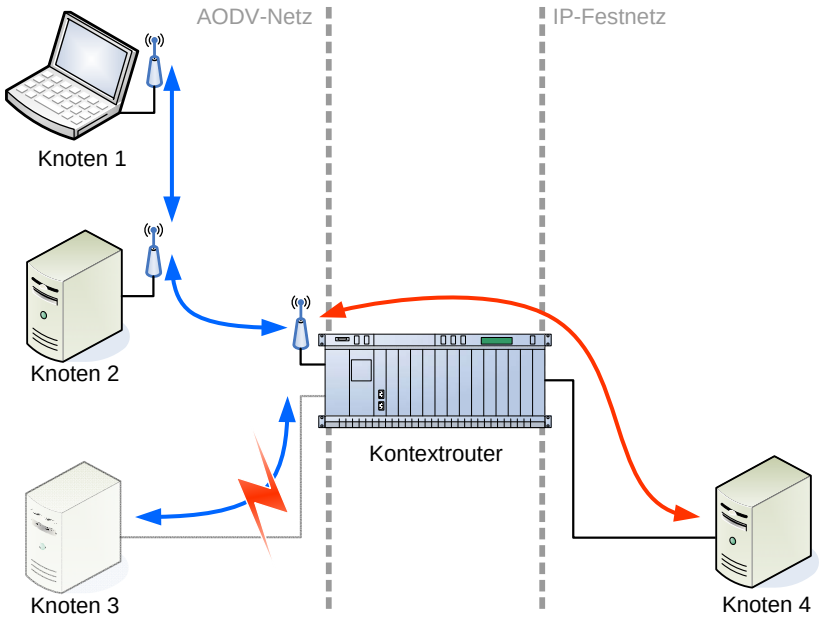

Abbildung 5.14: Aufbau Testszenario 3

Das Programm `cont_client` startet nach Abschluss der Dienstsuche das bereits besprochene Programm `CIPPING` selbstständig. Hierbei werden die vom Kontextrouter erhaltenen Parameter `ID` und `Session` übergeben. Diese beiden Parameter sind notwendig, um die in Kapitel 2.3 beschriebenen IP-Optionen zu erstellen.

Der Kontextrouter nutzt die gesetzten IP-Optionen, um eintreffende IP-Pakete an den zuvor ermittelten Dienstanbieter weiterzuleiten und umgekehrt. Die blauen Pfeile in Abbildung 5.14 symbolisieren den zurückgelegten Weg der Anwendungsdatenpakete.

Nach kurzer Zeit wird jedoch die Verbindung zum ersten angebotenen Dienstanbieter (*Knoten 3*) getrennt. Im Anschluss ist es Aufgabe des Kontextrouters, diese Verbindungsunterbrechung zu detektieren und den Anwendungsdatenstrom zu einem geeigneten Dienstanbieter umzuleiten. Die durchgeführte Umleitung bleibt vom Client unbemerkt.

Der Client sucht wie im Testszenario zwei auch nach Diensttyp „`30`“ und den in Tabelle 5.7 abgebildeten Kontexttypen und Prioritäten. Weiterhin ist die Liste der unerwünschten Dienstanbieter in diesem Szenario leer. Somit sind alle Dienstanbieter vom Client zu akzeptieren. Die Dienstregistrierung unterscheidet sich jedoch geringfügig vom Szenario 2. Tabelle 5.9 zeigt die am Kontextrouter registrierten Dienste. Somit ist gewährleistet, dass eine Anfrage nach Dienst „`30`“ mit der in Tabelle 5.7 gezeigten Kontexttypen und Prioritäten *Knoten 3* (`192.168.2.50`) als ersten Dienstanbieter liefert. Da sich dieser Knoten im AODV-Netz befindet, kann ein Abreißen der Verbindung rasch erkannt werden, was das *Rerouting* der Anwendungsdaten erleichtert.

IP-Adresse	**Diensttyp**	**Kontexttypen**
192.168.2.50	30	1,4,5,7,8
192.168.1.30	30	1,2,3

Tabelle 5.9: Registierte Dienste – Testszenario 3

Abbildung 5.15 zeigt die via `cont_client` getätigten Eingaben und dessen Rückmeldungen. Wie zu sehen ist, wird das Programm `CIPPING` mit den Parametern `-n -i2 -c20 -C 1_192.168.2.50 10.0.0.10` aufgerufen. Wobei das Argument `-c20 CIPPING` veranlasst, nach zwanzig gesendeten *ICMP*-Paketen zu terminieren. Das bereits beschriebene Argument `-C` übergibt `Session` und `ID`-Parameter an das Programm. Die `CIPPING`-Ausgabe „`Contxet IP option ...`“ erscheint, sobald ein empfangenes Anwortpaket die beschriebene IP-Option enthält. Weiterhin sind lange Programmausgaben zur besseren Lesbarkeit umgebrochen und mit dem Zeichen „\“ gekennzeichnet.

Abbildung 5.16 zeigt die durchgeführte Dienstsuche und einige der anschließend versandten *ICMP echo request* und -*reply*-Pakete. Aufgrund der erzielten Ergebnisse des

```
# ./cont_client localhost
Nachricht zum Versenden: context-request router=10.0.0.10 service=30 \
  ct=1,7:2,5:3,4:4,6:5,3:6,2:7,2:8,3:9,4:10,0
MSG from routing daemon: ACK ++ requesting context service...

MSG from routing daemon: Service-reply from Router=10.0.0.10, Service=30, \
  ID=192.168.2.50, Session=1, CT=1,4,5,7,8, NA=0

calling: ./cipping -n -i2 -c20 -C 1_192.168.2.50 10.0.0.10
CIPPING 10.0.0.10 (10.0.0.10) 56(84) bytes of data.
64 bytes from 10.0.0.10: icmp_seq=1 ttl=254 time=83.3 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)

...

64 bytes from 10.0.0.10: icmp_seq=7 ttl=254 time=65.8 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)
64 bytes from 10.0.0.10: icmp_seq=8 ttl=254 time=61.7 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)
64 bytes from 10.0.0.10: icmp_seq=10 ttl=254 time=7683 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)
64 bytes from 10.0.0.10: icmp_seq=14 ttl=254 time=83.0 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)
64 bytes from 10.0.0.10: icmp_seq=15 ttl=254 time=82.7 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)
64 bytes from 10.0.0.10: icmp_seq=16 ttl=254 time=79.7 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)

...

--- 10.0.0.10 ping statistics ---
20 packets transmitted, 16 received, 20% packet loss, time 38011ms
rtt min/avg/max/mdev = 61.744/548.741/7683.566/1842.215 ms, pipe 4
Nachricht zum Versenden: quit
MSG from routing daemon: ACK -- Quitting...
```

Abbildung 5.15: `cont_client`-Bildschirmausgaben – Testszenario 3

vorherigen Testszenario kann auf eine Überprüfung der versendeten und empfangenen *CRREQ* und *CRREP*-Pakete verzichtet werden.

Abbildung 5.17 zeigt den Paketaufbau des ersten versandten *ICMP echo request*-Paketes durch das Programm `CIPPING`. Diese Nachricht trägt in Abbildung 5.16 die Paketnummer `23`. Die enthaltene IP-Option ist blau markiert. Wie zu sehen ist, enthält sie nach Typen- und Längenangaben den übermittelten `Session`- (`00 01`) sowie den `ID`-Parameter (`c0 a8 02 32`) in hexadezimaler Notation und *Network Byte Order*.

In Abbildung 5.18 ist das erste empfangene *ICMP echo repy*-Paket zu erkennen. Der blau hinterlegte Bereich markiert die enthaltene IP-Option. Anhand der identischen *Sequence number* im Bereich *Internet Control Message Protocol* der Abbildungen 5.17

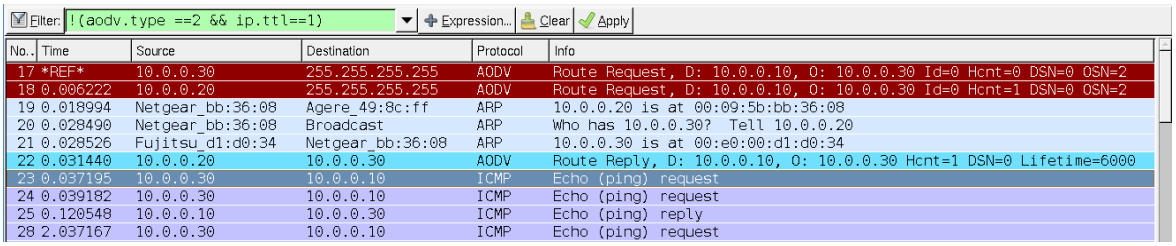

Abbildung 5.16: Wireshark Screenshot – Testszenario 3, Übersicht, vor Serverausfall

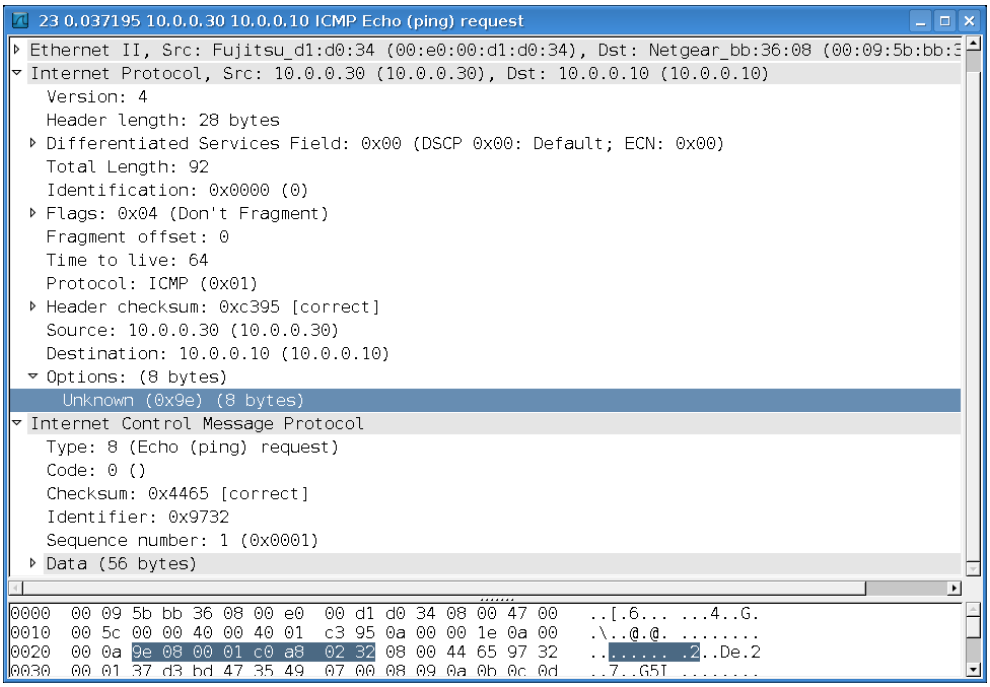

Abbildung 5.17: Wireshark Screenshot – Testszenario 3, *ICMP echo request*

und 5.18 ist die „Zugehörigkeit“ der beiden Pakete erkennbar. Weiterhin enthalten sowohl die gesendeten *echo request* als auch die empfangenen *echo reply*-Pakete identische IP-Optionen. Das in Abbildung 5.20 gezeigte Paket entspricht der `CIPPING`-Ausgabe aus Abbildung 5.15 mit den Parametern `icmp_seq=10 ttl=254 time=7683 ms`.

Nach kurzer Zeit wurde *Knoten 3* (`192.168.2.50`) manuell vom Kontextrouter getrennt. Somit ist dieser gezwungen, einen alternativen Dienstanbieter für die bestehende Anwendungskommunikation zu finden. In Abbildung 5.19 ist dieser Zeitpunkt ersichtlich. Wie zu erkennen ist, versendet der Kontextrouter mehrere *RREQ*-Pakete. Vor der Ermittlung eines Alternativdienstanbieters suchen diese Pakete nach einer Route zum ausgefallenen Dienstanbieter *Knoten 3*.

In diesem Szenario findet der Kontextrouter keine neue Route zum bisherigen Dienstanbieter *Knoten 3*. Somit ermittelt der Kontextrouter *Knoten 4* (`192.168.1.30`)

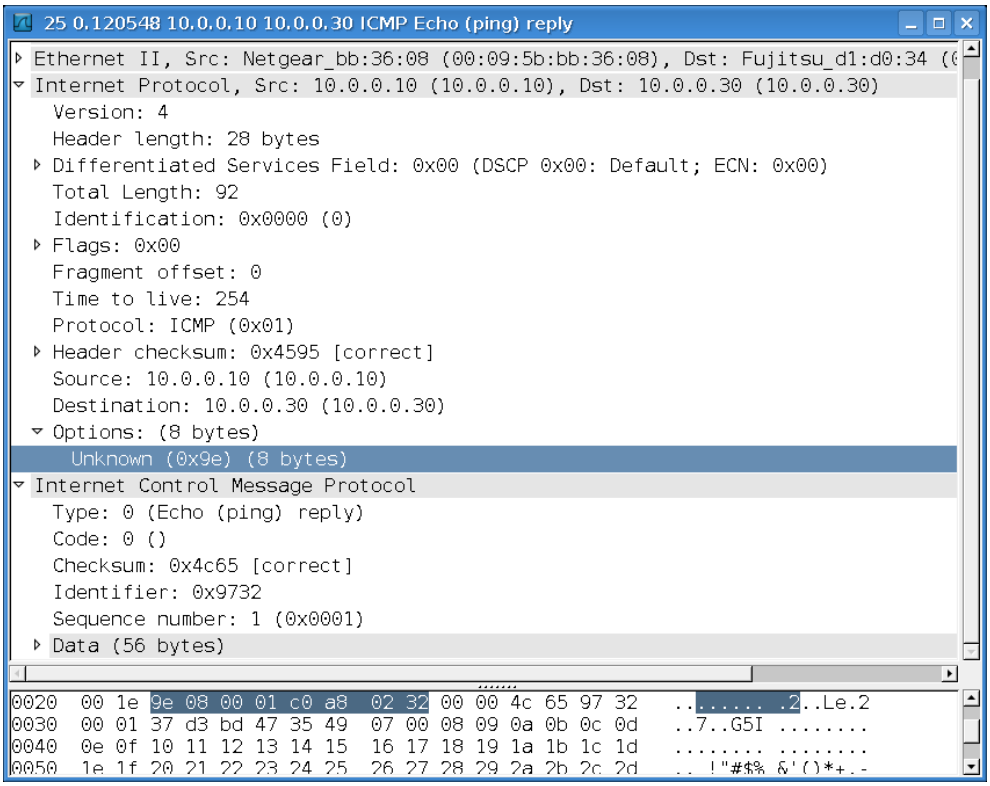

Abbildung 5.18: Wireshark Screenshot – Testszenario 3, *ICMP echo reply*

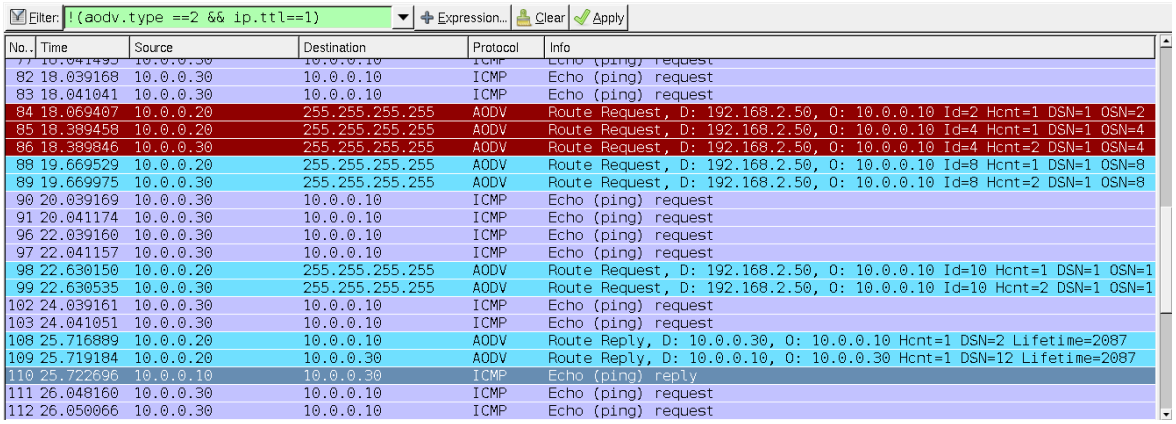

Abbildung 5.19: Wireshark Screenshot – Testszenario 3, Übersicht, Serverausfall

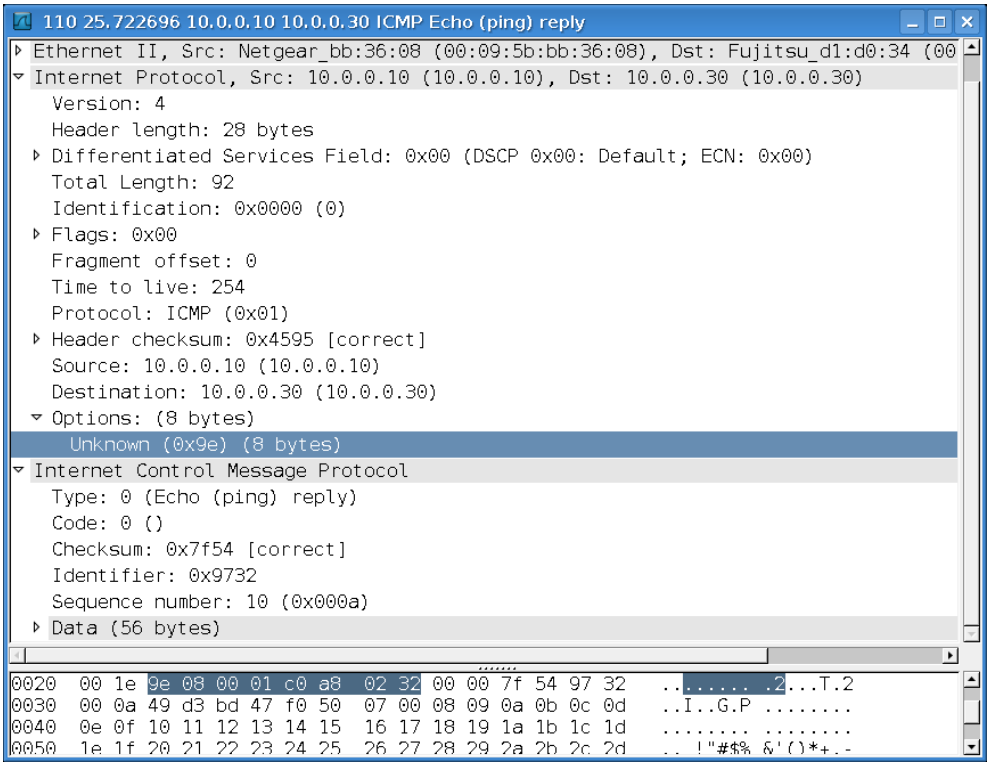

Abbildung 5.20: Wireshark Screenshot – Testszenario 3, *ICMP echo reply* nach Serverausfall

als alternativen Dienstanbieter und leitet fortan alle eintreffenden IP-Pakete mit der beschriebenen IP-Option an *Knoten 4* und umgekehrt weiter.

Abbildung 5.20 zeigt das erste *ICMP echo reply*-Paket vom neu ermittelten Dienstanbieter. Das Paket trägt die Nummer 110 in Abbildung 5.19. Der blau markierte Bereich stellt die enthaltene IP-Option dar. Wie zu erkennen, ist die übermittelte IP-Option mit den vorherigen versandten IP-Optionen identisch. Somit ist das erfolgte *Rerouting* auf einen neuen Dienstanbieter für den Client nicht ersichtlich.

Es ist anzunehmen, dass der aus Abbildung 5.15 ersichtliche Paketverlust aufgrund des Serverausfalls und dem anschließenden *Rerouting* eintritt. Hierdurch hat die Anwendung ein Indiz für einen Serverausfall. Dies stellt aber keinen sicheren Beweis dar. Es bleibt jedoch Aufgabe der Anwendung, eintretende Verzögerungen oder Paketverluste, wie in Abbildung 5.15 zu sehen ist, selbstständig abzufangen oder zu kompensieren. Eine solche Kompensation von Paketverlusten ist eng vom genutzten Dienst und dessen verwendeten Übertragungsprotokollen abhängig. Somit kann bezüglich der unterbrechungsfreien Dienstnutzung im *Rerouting*-Fall keine allgemein gültige Aussage getroffen werden.

```
# ./cont_client localhost
Nachricht zum Versenden: context-request router=10.0.0.10 service=30 \
  ct=1,7:2,5:3,4:4,6:5,3:6,2:7,2:8,3:9,4:10,0 no-reroute
MSG from routing daemon: ACK ++ Rerouting disabled
ACK ++ requesting context service...

MSG from routing daemon: Service-reply from Router=10.0.0.10, Service=30, \
  ID=192.168.2.50, Session=1, CT=1,4,5,7,8, NA=0

calling: ./cipping -n -i2 -c20 -C 1_192.168.2.50 10.0.0.10
CIPPING 10.0.0.10 (10.0.0.10) 56(84) bytes of data.
64 bytes from 10.0.0.10: icmp_seq=1 ttl=254 time=59.8 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)

...

64 bytes from 10.0.0.10: icmp_seq=7 ttl=254 time=66.7 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)
64 bytes from 10.0.0.10: icmp_seq=9 ttl=254 time=7691 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)
64 bytes from 10.0.0.10: icmp_seq=13 ttl=254 time=84.4 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)
64 bytes from 10.0.0.10: icmp_seq=14 ttl=254 time=82.3 ms
Contxet IP option, session: 1, id: 192.168.2.50 (3232236082)

...

--- 10.0.0.10 ping statistics ---
20 packets transmitted, 16 received, 20% packet loss, time 38000ms
rtt min/avg/max/mdev = 59.862/548.496/7691.307/1844.278 ms, pipe 4
Nachricht zum Versenden: quit
MSG from routing daemon: ACK -- Quitting...
```

Abbildung 5.21: `cont_client`-Bildschirmausgaben – Testszenario 3, mit *No Reroute*-Flag

Der zweiten Teil dieses Testszearios soll die Wirkung des neu eingeführten *No Reroute*-Flag demonstrieren. Die Voraussetzungen gleichen denen des ersten Teils dieses Szenarios. So sind die in Tabelle 5.9 gezeigten Dienste am Kontextrouter registriert. Weiterhin wird nach der Dienstnummer „30“ mit den in Tabelle 5.8 abgebildeten Kontexttypen und Prioritäten gesucht.

Abbildung 5.21 zeigt den zugehörigen `cont_client`-Aufruf und die resultierenden `CIPPING`-Ausgaben. Zu erkennen ist, dass sich im Vergleich zu Abbildung 5.15 der abgesetzte `context-request`-Befehl lediglich durch das angehängte Schlüsselwort `no-reroute` unterscheidet.

Der Ablauf dieses Testes unterscheidet sich bis auf das gesetzte *No Reroute*-Flag nicht von den vorherigen Tests. Kurz nach Start des `CIPPING`-Programms wurde *Kno-*

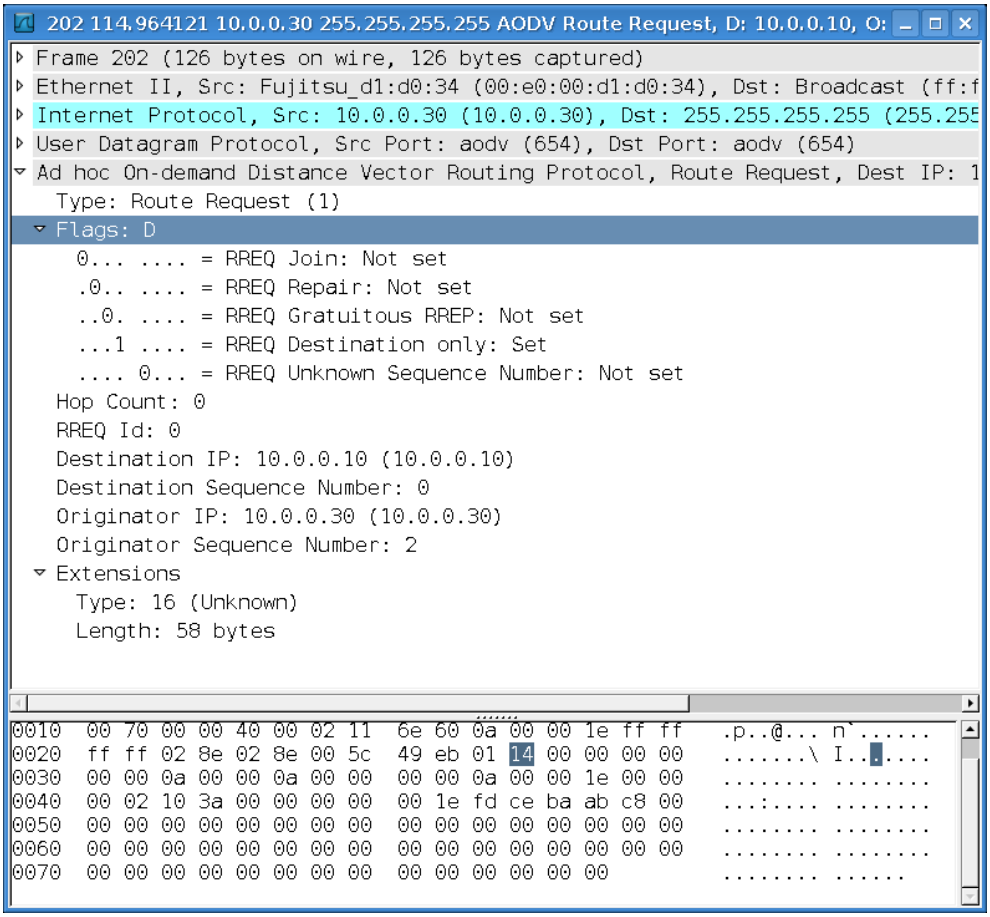

Abbildung 5.22: Wireshark Screenshot – Testszenario 3, *CRREQ* mit *No Reroute*-Flag

ten 3 manuell aus dem AODV-Netzwerk entfernt. Dies geschah durch simples deaktivieren des AODV-Daemons.

Theoretisch sollte nach dieser Unterbrechung der Anwendungskommunikation der Kontextrouter keinen alternativen Dienstanbieter bestimmen. Die in Abbildung 5.21 gezeigten `CIPPING`-Ausgaben weisen ein anderes Verhalten nach. So führt der Kontextrouter trotz aktiviertem *No Reroute*-Flag ein *Rerouting* nach Erkennung des Serverausfalls durch.

Abbildung 5.22 zeigt das anfangs versandte *CRREQ*-Paket mit aktivem *No Reroute*-Flag. Der blau markierte Bereich kennzeichnet die Flags des Paketes. Weiterhin ist die enthaltene Kontexterweiterung identisch mit dem in Abbildung 5.9 gezeigten *CRREQ*-Paket. Somit ist belegt, dass das *CRREQ*-Paket korrekt erstellt wurde und ein Fehler im Kontextrouter vorzuliegen scheint.

Die vollständigen *Wireshark-capture Files*, die `cont_client`-Bildschirmausgaben sowie die komplette AODV-UU-Log Dateien sind dem beiliegenden Datenträger aus Ver-

zeichnis `/messungen/Szenario3` zu entnehmen.

5.2.4 Szenario 4: Variation des Time-outs von erweiterten RREQ Nachrichten

Dieses Testszenario soll das Variieren des Time-Out-Parameters zum Versenden der *CRREQ*-Nachrichten demonstrieren. Der Parameter lässt sich mittels dem `crreq-timeout`-Schlüsselwort beeinflussen. Eine Übermittlung des Schlüsselwortes kann mit Hilfe des Programms `cont_client` an AODV-UU erfolgen.

Weiterhin entspricht die Dienstsuche Testszenario zwei. Die versandten *CRREQ*-Pakete suchen nach Diensttyp „`30`“ und der in Tabelle 5.7 angegebenen Kontexttypen und Prioritäten. Die Liste unerwünschter Dienstanbieter (*Blacklist*) ist leer.

Das Überschreiten der Time-out-Werte wird erzwungen, indem *Knoten 1* keinen anderen Knoten der Demonstratorumgebung erreicht. Somit versendet *Knoten 1* die CRREQ-Nachricht einige Male, bevor eine Meldung über das Misslingen der Dienstsuche ausgegeben wird.

Im ersten Test soll der Time-out-Wert `200` Millisekunden betragen. Abbildung 5.23 zeigt die getätigten Eingaben sowie die resultierenden Meldungen des AODV-Daemons. Im Wireshark-Protokollanalysator ist aus Abbildung 5.24 zu erkennen, dass der Abstand der versandten *CRREQ*-Pakete nahezu 200 Millisekunden beträgt. Hierbei dient das erste versendete *CRREQ*-Paket, welche in Abbildung 5.24 die Nummer fünfzehn trägt, als Zeitreferenz.

Im Anschluss wurde nach Abbildung 5.25 der *CRREQ*-Time-Out-Parameter auf `2000` Millisekunden erhöht. Wie der Wiresharkausgabe aus Abbildung 5.26 zu entnehmen ist, erfolgt das Versenden der *CRREQ*-Nachrichten alle `2000` Millisekunden.

Die Abbildungen 5.23 bis 5.26 zeigen, dass der neu erstellte Parameter zur Manipulation der Time-out-Werte beim Versand von *CRREQ*-Nachrichten die gewünschte Wirkung erzielt.

Die vollständigen *Wireshark-capture Files*, die `cont_client`-Bildschirmausgaben, sowie die komplette AODV-UU-Log Dateien sind dem beiliegenden Datenträger aus Verzeichnis `/messungen/Szenario4` zu entnehmen.

```
# ./cont_client localhost
Nachricht zum Versenden: crreq-timeout=200
MSG from routing daemon: ACK -- context RREQ Timeout was set to: 200
To disable user context RREQ-timeout, set it to zero.

Nachricht zum Versenden: context-request router=10.0.0.10 service=30 \
  ct=1,7:2,5:3,4:4,6:5,3:6,2:7,2:8,3:9,4:10,0
MSG from routing daemon: ACK ++ requesting context service...

MSG from routing daemon: Sorry, couldn't reach the context router (10.0.0.10)!
Please try to use another context router with router=<IP> keyword or wait for\
  a Router adversisement Message!
Quitting...
```

Abbildung 5.23: `cont_client`-Bildschirmausgaben – Testszenario 4, Time-out 200 ms

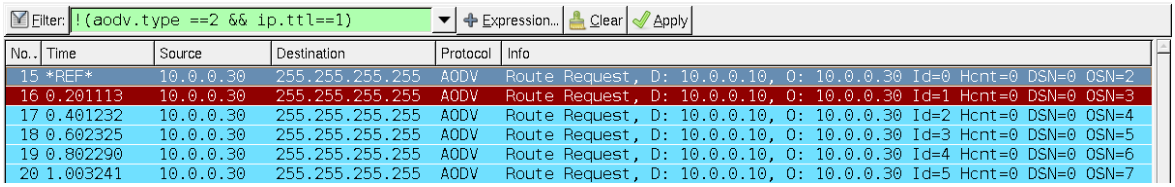

Abbildung 5.24: Wireshark Screenshot – Testszenario 4, Übersicht, CRREQ Time-out 200 ms

```
# ./cont_client localhost
Nachricht zum Versenden: crreq-timeout=2000
MSG from routing daemon: ACK -- context RREQ Timeout was set to: 2000
To disable user context RREQ-timeout, set it to zero.

Nachricht zum Versenden: context-request router=10.0.0.10 service=30 \
  ct=1,7:2,5:3,4:4,6:5,3:6,2:7,2:8,3:9,4:10,0
MSG from routing daemon: ACK ++ requesting context service...

MSG from routing daemon: Sorry, couldn't reach the context router (10.0.0.10)!
Please try to use another context router with router=<IP> keyword or wait for\
  a Router adversisement Message!
Quitting...
```

Abbildung 5.25: `cont_client`-Bildschirmausgaben – Testszenario 4, Time-out 2000 ms

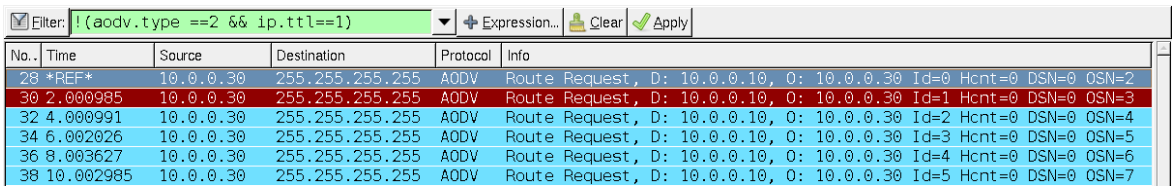

Abbildung 5.26: Wireshark Screenshot – Testszenario 4, Übersicht, CRREQ Time-out 2000 ms

5.2.5 Szenario 5: Kontextsensitive Dienstsuche durch Festnetzclient

Die bereits beschriebene Festnetzvariante der erstellten AODV-UU-Implementierung soll in gewöhnlichen IP-Netzen zum Einsatz kommen, in denen kein AODV zur Routensuche verwendet wird. Somit ist es möglich, die genannten Dienstanfragen in herkömmlichen Netzwerken durchzuführen.

Hierbei dient *Knoten 4*, der den via Ethernet verbundenen Kontextrouter kontaktiert und eine *CRREQ*-Nachricht übermittelt, als Client. Abbildung 5.27 zeigt den schematischen Aufbau des Testszenarios. Anschließend soll der Kontextrouter, wie in den vorhergehenden Testszenarien auch, einen passenden Dienst mittels einer *CRREP*-Nachricht anbieten.

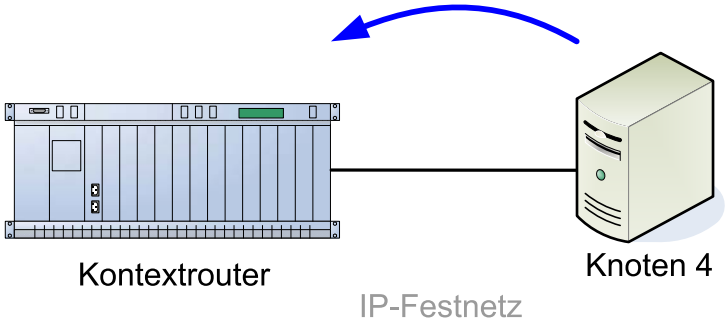

Abbildung 5.27: Aufbau Testszenario 5

Der gesuchte Dienst entspricht den Parametern aus Testszenario zwei. Die versandten *CRREQ*-Pakete suchen nach Diensttyp „30“ und der in Tabelle 5.7 angegebenen Kontexttypen und Prioritäten. Die Liste unerwünschter Dienstanbieter (*Blacklist*) ist leer. Anzumerken ist weiterhin, dass der AODV-UU-Festnetzclient zwingend mit der Option `-i ethX` zu starten ist. Wobei `ethX` das verwendete Netzwerkgerät spezifiziert. Wird dieser Parameter nicht angegeben, versucht AODV-UU sich an das erste verfügbare *WLAN*-Interface zu binden.

Abbildung 5.28 zeigt die getätigten Eingaben sowie die entsprechenden Rückmeldungen des AODV-Daemons. Wie leicht zu erkennen, ist die abgesetzte Dienstsuche nicht erfolgreich. Abbildung 5.29 stellt die mittels Wireshark aufgezeichneten Pakete dar. Die ersten drei gezeigten Pakete sind die vom Kontextrouter versandten *ICMP router advertisement*-Nachrichten. Somit ist eine generelle Kommunikationsstörung zwischen *Knoten 4* und dem Kontextrouter auszuschließen.

In der Wiresharkausgabe aus Abbildung 5.29 ist kein Filter zu erkennen, somit sind

```
# ./cont_client localhost
Nachricht zum Versenden: context-request router=192.168.1.10 service=30 \
  ct=1,7:2,5:3,4:4,6:5,3:6,2:7,2:8,3:9,4:10,0
MSG from routing daemon: ACK ++ requesting context service...

MSG from routing daemon: Sorry, couldn't reach the context router (192.168.1.10)!
Please try to use another context router with router=<IP> keyword or wait for a \
  Router adversisement Message!
Quitting...
```

Abbildung 5.28: `cont_client`-Bildschirmausgaben – Testszenario 5

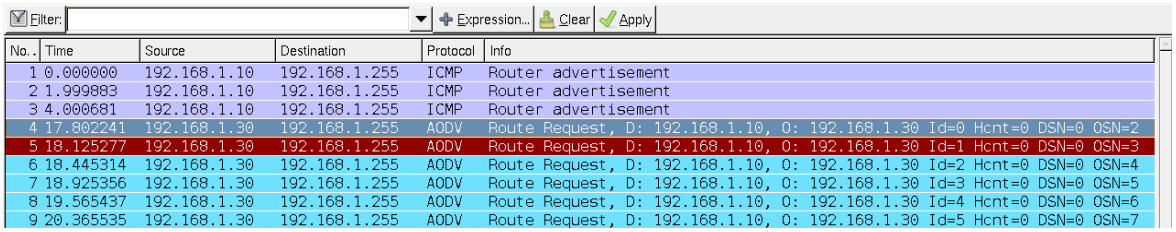

Abbildung 5.29: Wireshark Screenshot – Testszenario 5, Übersicht

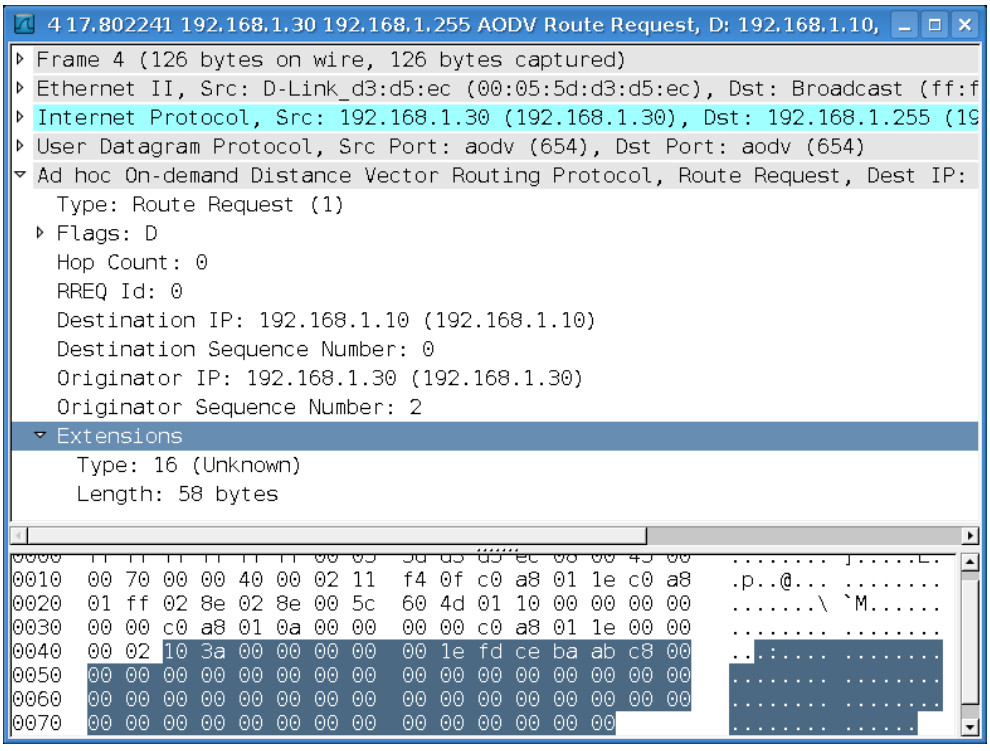

Abbildung 5.30: Wireshark Screenshot – Testszenario 5, CRREQ

alle gesendeten und empfangenen Pakete abgebildet. AODV versendet im Normalfall pro Sekunde eine *Hello*-Nachricht. Diese Nachrichten dienen der Aktualisierung einer Liste von Nachbarknoten. Nach Abbildung 5.29 versendet die Festnetzvariante der AODV-UU-Implementierung jedoch keine *Hello*-Nachrichten. Dieses Verhalten ist korrekt, da es sich nicht um ein AODV-Netzwerk handelt und hierdurch auch keine zyklischen *Hello*-Nachrichten benötigt werden.

Anschließend sind in Abbildung 5.29 die vom AODV-UU-Festnetzclient versandten *CRREQ*-Pakete erkennbar. Wie im Testszenario vier wiederholt der AODV-Daemon das Versenden der *CRREQ*-Pakete nach Verstreichen eines Time-out-Wertes einige Male.

Abbildung 5.30 enthält die erste versandte *CRREQ*-Nachricht. Wie zu erkennen ist, entspricht der Aufbau des gezeigten Paketes sowie der enthaltenen Kontexterweiterung den in vorherigen Testszenarien abgebildeten *CRREQ*-Paketen. Somit kann von einer ordnungsgemäßen Erstellung der CRREQ-Pakete ausgegangen werden.

Nach Prüfung des Paketempfangs am Kontextrouters sowie Rücksprache mit dem Autor der click-Erweiterungen Marco Wenzel konnte das Problem auf einen Fehler im click-Element `GenerateCRREP` eingegrenzt werden. Jedoch war es trotz intensiver Bemühungen bis Fertigstellung dieser Arbeit nicht möglich, das Problem zu beseitigen.

Somit belegt dieses Testszenario, dass keine *Hello*-Nachrichten versandt werden und dass die erstellten CRREQ-Pakete gemäß der Spezifikation aufgebaut sind.

Das vollständige *Wireshark-capture File*, die `cont_client`-Bildschirmausgabe sowie die komplette AODV-UU-Log Datei sind dem beiliegenden Datenträger aus Verzeichnis `/messungen/Szenario5` zu entnehmen. Weiterhin ist ein *Wireshark capture File* des Kontextrouters enthalten, welches das Eintreffen der von *Knoten 4* versandten Pakete zeigt.

6 Ausblick

Konzeption und Umsetzung der im Rahmen dieser Arbeit angefertigten und modifizierten Programme erfolgte unter dem Aspekt einer möglichst flexiblen Nutzung. Wobei eine eventuelle Weiterentwicklung oder Anpassung der erstellten Anwendungen berücksichtigt wurde. Neben der Dokumentation von der im Rahmen dieser Arbeit erstellten oder veränderten Programmteile enthalten die jeweiligen Quelltexte zahlreiche Kommentare. Dies soll die Lesbarkeit und Einarbeitung in den Programmcode erleichtern, so dass spätere Änderungen leicht umzusetzen sind.

Daneben stellt die Implementierung des in Kapitel 4.3.3 vorgestellten alternativen Konzeptes einer transparenten Anwendungsschnittstelle eine große Herausforderung dar. Mit der Umsetzung dieser Schnittstelle wäre ein weiterer Schritt in Richtung einer transparenten Nutzung kontextbehafteter Dienste vollzogen. Eine Anpassung aller vom Client verwendeten Anwendungen zur Dienstnutzung ist oftmals nicht möglich oder nur äußerst aufwendig zur realisieren.

Die Umsetzung der genannten Schnittstelle zur kontextsensitiven Dienstnutzung wirft weitere Fragen auf. Es existiert nach Kenntnissstand des Autors kein Konzept, welches die reale Dienstnutzung spezifiziert. Daher sind vor der eigentlichen Dienstnutzung die gewünschte Dienstnummer sowie die unterstützen Kontexttypen zu übergeben. Kapitel 4.3.3 schlägt hierfür zum Beispiel eine *Hilfsanwendung* vor. Diese kennt die vom jeweiligen Nutzer unterstützten Kontexttypen und initiiert die Dienstsuche. Im Anschluss startet diese Anwendung das eigentliche Anwendungsprogramm. Das beschriebene Beispiel hätte den Vorteil, dass die genutzten Anwendungsprogramme keinerlei Kenntnisse über die verwendete Kontextroutingarchitektur benötigen. Dieser Prozess bedarf jedoch weiterer Diskussion.

Ein weiterer offener Aspekt stellt der Mangel an real existierenden Dienstanbietern dar, welche die beschriebenen Protokolle unterstützen. So wurde im Rahmen der Verifizierung der erstellten Programme im Kapitel 5.2 dieser Arbeit ein modifizierter `ping`-Dienst verwendet. Dieser Dienst beantwortet schlicht die eintreffenden *ICMP echo request*-Nachrichten, die mit der beschriebenen IP-Option versehen sind.

Ein solcher Dienst mag für akademische Zwecke ausreichend erscheinen. Jedoch sind

die Ansprüche gewöhnlicher Nutzer in der Regel etwas anders ausgeprägt. So sollten mögliche *kontextsensitive* Dienste einen zu erkennenden Mehrwert gegenüber *kontextlosen* Diensten für den Nutzer bieten.

Die Frage einer Zuordnung von realen Diensten und Kontexttypen zu den bisher verwendeten abstrakten Nummern ist momentan ebenfalls ungeklärt. Diese Assoziation spielt auf der technischen Ebene der realisierten Architektur keine Rolle. Vielmehr ist dies ein Problem der praktischen Nutzung von kontextsensitiven Diensten. So bietet sich eine zentrale Verwaltungsinstanz an, die konkreten Dienst- und Kontexttypen den bestehenden Nummern zuweist und verwaltet.

Daneben sind noch weitere technische Verbesserungen vorstellbar. Die Multicastadressierung ist momentan weder vom bestehenden Kontextrouter noch von der im Rahmen dieser Arbeit entwickelten AODV-UU-Modifizierung nutzbar. AODV bietet bereits grundlegende Unterstützung von Multicastadressierung. Allerdings muss die jeweilige AODV-Implementierung zusätzliche, in [RP00] beschriebene, Tabellen verwalten. Für AODV-UU existiert mit [7] eine Multicasterweiterung, die jedoch AODV-UU-Version 0.6 oder 0.7.2 voraussetzt. Diese Erweiterung ist nicht mit der aktuellen AODV-UU-Version 0.9.5 kompatibel, da im Laufe der AODV-UU-Entwicklung zahlreiche Änderungen an der internen Struktur vorgenommen wurden. Diese Erweiterung bildet jedoch einen Ansatzpunkt einer zukünftigen an die aktuelle AODV-UU-Version angepasste Multicastimplementierung.

Aufgrund mangelnder Multicastfähigkeit versendet der beschriebene click-Kontextrouter *ICMP router advertisement*-Nachrichten im Broadcastverfahren, um seine Adresse im Netzwerk bekannt zu machen. Gewöhnliche Netzwerkknoten leiten derartige Broadcastnachrichten nicht weiter. Eine Weiterleitung aller Broadcastpakete birgt jedoch die Gefahr einer Überflutung und Blockierung des lokalen Netzwerkes. [Wen07b] begrenzt mittels des TTL-Parameters im IP-Header die Reichweite der versandten *ICMP router advertisement*-Nachrichten auf wenige Hops. Somit würde eine selektive Weiterleitung dieser Nachrichten das umgebende Netzwerk nur unter bestimmten Voraussetzungen blockieren. Die vorgestellte AODV-UU-Erweiterung könnte weiterhin um diese selektive Broadcastweiterleitungsfunktion ergänzt werden.

Die Variation von Protokollparametern ist speziell für ausgiebige Testszenarien interessant. AODV-UU fasst nahezu alle vom RFC 3561 [PBRD03] vorgeschlagenen Protokollparameter in der Datei `params.h` zusammen. Hierbei kommen *C-Präprozessorkonstanten* zum Einsatz. Es wäre weiterhin denkbar, diese Parameter während der Laufzeit zu beeinflussen. In diesem Fall müssen die *Präprozessorkonstanten* auf Pro-

grammvariablen verweisen. Somit ließe sich ohne umfangreiche Änderungen der Programmstrukturen ein flexibles Variieren der AODV-Protokollparameter realisieren. Die Parameter `ACTIVE_ROUTE_TIMEOUT`, `TTL_START` und `DELETE_PERIOD` sind bereits durch AODV-UU variabel konzipiert.

Die im Rahmen dieser Arbeit erwähnten Unstimmigkeiten in der click-Implementierung des Kontextrouters schränken einige Funktionen der Demonstratorumgebung ein. Um eine möglichst flexible Testumgebung zu erhalten, ist eine Beseitigung dieser Einschränkungen unumgänglich.

7 Zusammenfassung

In der vorliegenden Diplomarbeit wurde die am Fachgebiet Kommunikationsnetze der Technischen Universität Ilmenau entwickelte Demonstratorumgebung zum kontextsensitiven Routing durch weitere Komponenten ergänzt. Diese Demonstratorumgebung ermöglicht es erstmals, die entworfenen Protokolle und Verfahren unter praxisnahen Bedingungen zu verifizieren.

Zuvor fand eine Analyse der technischen Anforderungen zur Realisierung von reaktiven Routingprotokollen statt. Dabei wurde auf real existierende Implementierungsansätze eingegangen, die es ermöglichen, auf die beschriebenen Anforderungen zu reagieren.

Das reaktive AODV-Routingprotokoll bildet die Grundlage der entwickelten Verfahren zum kontextsensitiven Routing. Daher wurden bestehende AODV-Implementierungen untersucht und verglichen. Der Anspruch einer möglichen Erweiterung um die beschriebenen Protokolle zur Nutzung kontextsensitiver Dienste spielte die maßgebliche Rolle.

Nach gründlicher Abwägung der jeweiligen Vor- und Nachteile der präsentierten AODV-Implementierungen wurde AODV-UU der schwedischen Universität in Uppsala für die anstehende Realisierung der Clientsoftware ausgewählt.

Das Konzept zur Einbindung in die Demonstratorumgebung sah unter anderem die Umsetzung der anfangs beschriebenen Protokollerweiterungen vor. Erst diese ermöglichen dem Client eine Übermittlung von kontextbehafteten Dienstanfragen und -antworten an den bereits realisierten Kontextrouter. Zur Nutzung der umgesetzten Protokollerweiterungen ist eine zusätzliche Schnittstelle zu kontextsensitiven Anwendungen notwendig. Das entworfene Schnittstellenkonzept sowie die reale Implementierung erlauben die manuelle Übergabe von Dienstanfragen. Daneben ist eine automatische Nutzung der Schnittstelle durch zukünftige kontextsensitive Anwendungsprogramme nicht ausgeschlossen. Weiterhin eignet sich die vorgestellte Schnittstelle zur Manipulation von Protokoll- und Funktionsparametern.

Ein Verifizieren der erfolgten Kommunikationsprozesse kann anhand von aussagekräftigen Debuggausgaben der erstellten AODV-UU-Modifizierung erfolgen. Daneben

ist der Einsatz von gewöhnlichen Netzwerk- und Protokollanalysatoren möglich. Diese bieten unter anderem eine zeitversetze Kontrolle der gesendeten und empfangen Daten an. Hierzu wurde der frei verfügbare Protokollanalysator *Wireshark* eingesetzt.

In den durchgeführten Funktionstests konnte am Demonstrator die fehlerfreie Funktionsweise der realisierten Programme gezeigt werden. Dazu wurden verschiedene Testszenarien entworfen, durchgeführt und anschließend ausgewertet. Somit war die Verifikation aller notwendigen Schritte des kontextsensitiven Kommunikationsprozesses möglich.

Aus Mangel an bestehenden Anwendungen und Dienstanbietern zur Nutzung der beschriebenen Architekturen kamen sowohl selbst erstellte als auch im Rahmen anderer studentischer Arbeiten realisierte Hilfsprogramme zum Einsatz. So wurde durch das Senden und Empfangen von modifizierten *ICMP echo*-Nachrichten eine Nutzung kontextsensitiver Dienste nachgebildet.

Weiterhin stellten sich bei Tests der Demonstratorumgebung einige Einschränkungen der Kontextrouterimplementierung heraus. So war es zum Beispiel nicht möglich, Antworten auf kontextbehaftete Dienstanfragen aus dem *IP-Festnetz* zu erhalten. Nach intensiver Recherche und Kontaktierung des Autors der Kontextrouterimplementierung stellte sich heraus, dass die beschriebenen Probleme durch nachträglich ausgeführte Verbesserungen der Routersoftware hervorgerufen worden. Allerdings war es trotz intensiver Bemühungen nicht möglich, alle gefundenen Einschränkungen der Kontextrouterimplementierung zu beheben.

Die realisierten Programme setzten das am Anfang dieser Arbeit erwähnte dreischichtige Modell zur Nutzung kontextsensitiver Dienste teilweise um. Es erfolgt momentan noch keine konsequente Trennung zwischen der nutzenden Anwendung und der beschriebenen Zwischenschicht zur Umsetzung aller notwendigen Protokolle. Daher hat zum Beispiel die jeweilige Anwendung zur Zeit noch selbst Sorge zu tragen, dass die versendeten IP-Pakete die beschriebenen IP-Optionen enthalten.

Zuletzt wird mit der Vorstellung eines Alternativkonzeptes eine Möglichkeit präsentiert, die entworfene Schichtentrennung zur Nutzung kontextsensitiver Dienste in die Praxis umzusetzen.

Anhang

A Kontextclient-Software

A.1 Neue Funktionen

Im folgenden Abschnitt sind alle neu erstellten Funktionen aufgelistet. Hierbei wird deren Aufgabe kurz erläutert und Informationen zu den erwarteten und zurückgegebenen Parametern gegeben.

aodv_cont_ext_init – erstellt und initialisiert den Serversocket für anfragende kontextsensitive Anwendungen.

Argumente:	ohne
Rückgabewert:	ohne
Datei:	cont_ext.c, cont_ext.h

Diese Funktion wird während der Initialisierungsphase zu Beginn des Programms aufgerufen und erstellt einen TCP-Socket. Dieser Socket wartet auf Port `CONT_EXT_IF_PORT` auf eingehende Verbindunswünsche der Kontextanwendungen. Im Anschluss wird der erstellte Socket mit Hilfe der Funktion `attach_callback_func` zusammen mit der Behandlungsfunkiton `handle_new_cont_ext_app_sock` für den `select`-Aufruf in der Programmhauptschleife registriert. Durch den folgenden Aufruf der Funktion `c_blacklist_init` wird die Serverblackliste initiiert.

icmp_router_msg_init – erstellt und initialisiert einen *Raw Socket* für eingehende *ICMP router advertisement*-Nachrichten des Kontextrouters.

Argumente:	ohne
Rückgabewert:	ohne
Datei:	cont_ext.c, cont_ext.h

Die Funktion wird ebenfalls während des Initialisierungsprozesses aufgerufen. Sie erstellt einen *Raw Socket* für die Verarbetung eintreffender *ICMP router advertisement*-Nachrichten des Kontextrouters. Mittels eines `attach_callback_func`-Aufrufes wird die Funktion `get_icmp_router_msg` als Behandlungsfunktion zusammen mit dem erstellten Socket für den `select`-Aufruf in der Hauptschleife registriert.

get_icmp_router_msg – empfängt und analysiert eingehende *ICMP router advertisement*-Nachrichten des Kontextrouters.

Argumente: `int sock`
Rückgabewert: ohne
Datei: cont_ext.c

Nach einem `select`-Durchlauf der Hauptschleife wird diese Funktion aufgerufen, wenn neue ICMP-Pakete eingetroffen sind. Nach einer Überprüfung des Paketaufbaus wird die enthaltene Kontextrouteradresse und deren Lebensdauer für spätere Anfragen einer Anwendung gespeichert.

handle_new_cont_ext_app_sock – erstellt und registriert den Kommunikationssocket der TCP-Anwendungsschnittstelle.

Argumente: `int sock`
Rückgabewert: ohne
Datei: cont_ext.c

Der nach einem TCP-Verbindungsaufbau durch den `accept`-Aufruf erstellte Socket wird mittels der `attach_callback_func`-Funktion registriert. Hierbei übernimmt `handle_cont_ext_app_msg` die Aufgabe der Behandlungsfunktion.

send_cont_ext_app_msg – versendet Nachrichten über die TCP-Anwendungsschnittstelle.

Argumente: `int sock, char *buf`
Rückgabewert: `int`
Datei: cont_ext.c, cont_ext.h

Die Funktion versendet die übergebene Nachricht in `*buf` über den Socket `sock`.

send_icmp_soli – versendet eine *ICMP router solicitation*-Nachricht.

Argumente: `int sock`
Rückgabewert: `int`
Datei: cont_ext.c, cont_ext.h

Die Funktion versendet eine *ICMP router solicitation*-Nachricht über den angegebenen Socket `sock`.

handle_cont_ext_app_msg – empfängt und analysiert Nachrichten der TCP-Kontextanwendungsschnittstelle.

Argumente: `int sock`
Rückgabewert: ohne
Datei: cont_ext.c, cont_ext.h

Sobald von einer verbundenen Kontextanwendung neue Nachrichten eintreffen, wird diese Funktion nach dem `select`-Durchlauf der Programmhauptschleife aufgerufen. Dei erhaltene Nachricht wird mit Hilfe mehrerer verschachtelter `strtok_r`-Aufrufe zerlegt. Hierbei erfolgt eine Analyse der übermittelten Schlüsselwörter sowie deren Argumente. Auf die erkannten Schlüsselwörter und deren Argumente wird im Anschluss entsprechend reagiert. Die Reaktion umfasst eine versandte Bestätigungsnachricht über den angegebenen Socket `sock` und das Ausführen der angeforderten Aktion. So provozieren zum Beispiel die Schlüsselwörter `context-request` und `service` das Aufrufen der Funktion `c_rreq_route_discovery`, die eine kontextbehaftete Dienstanfrage einleitet. Somit stellt diese Funktion das zentrale Element der erstellten Schnittstelle dar, da hier auf eintreffende Befehle reagiert wird.

rreq_set_context – bearbeitet das Kontexttyp- und Prioritätenbitfeld der RREQ-Paketerweiterung zur Dienstsuche.

Argumente: `C_RREQ_EXT *c_ext, u_int8_t context_number, u_int8_t prio`
Rückgabewert: `int8_t`
Datei: cont_ext.c, cont_ext.h

Der übergebenen Wert des Kontexttyps (`context_number`) sowie dessen Priorität (`prio`) wird an die entsprechende Position der Protokollerweiterung geschrieben. Der angegebene Zeiger `c_ext` verweist dabei auf die entsprechende Datenstruktur.

c_rreq_send – erstellt und versendet eine RREQ-Nachricht mit angehängter Protokollerweiterung.

Argumente: `struct in_addr dest_addr, u_int32_t dest_seqno, int ttl, u_int8_t flags, C_RREQ_EXT *c_extension`
Rückgabewert: ohne
Datei: cont_ext.c, cont_ext.h

Diese Funktion basiert auf der von AODV-UU bereitgestellten Funktion `rreq_send` aus der Datei `aodv_rreq.c`. Im Gegensatz zur Orginalfunktion wird zusätzlich die durch `*c_extension` übergebene Protokollerweiterung an das erstellte RREQ-Paket angehangen und anschließend zum Versenden übergeben. Hierzu erstellt ein `rreq_create`-Aufruf das zugrundeliegende RREQ-Paket, danach fügt `rreq_add_ext`

die angegebene Erweiterung hinzu und die Funktion `aodv_socket_send` leitet letztendlich den Versand des erstellten Paketes ein.

c_rreq_route_discovery – leitet eine kontextbehaftete Dienstsuche ein.

Argumente:	`struct in_addr dest_addr, u_int8_t flags, C_RREQ_EXT * c_ext, int sock`
Rückgabewert:	ohne
Datei:	cont_ext.c, cont_ext.h

Diese Funktion basiert ebenfalls auf der von AODV-UU bereitgestellten Funktion `rreq_route_discovery` aus der Datei `aodv_rreq.c`. Es erfolgte jedoch eine Anpassung bezüglich der Behandlung von kontextbehafteten Dienstanfragen. Hierzu wird zu Beginn überprüft, ob bereits ein Eintrag in der Routingtabelle zur IP-Adresse des Kontextrouters (`dest_addr`) existiert. Ist dies der Fall, werden zum Beispiel die Sequenznummer des Zielknotens und der zuletzt verwendete ttl-Wert anstatt der Defaultwerte genutzt. Anschließend initiiert ein `c_rreq_send`-Aufruf das Versenden des Paketes.

Danach speichert ein Datensatz vom Typ `c_seek_list_t` alle relevanten Informationen, die zum einem für ein mögliches erneutes Versenden des Paketes benötigt werden. Zum anderen wird dieser Datensatz für die Bearbeitung einer möglichen Antwortnachricht benötigt. Zum Abschluss wird mittels der Funktion `timer_set_timeout` eine Time-out-Zeitspanne festgelegt. Wird innerhalb dieser Zeitspanne keine entsprechende Antwort empfangen, kommt die definierte Time-out-Behandlungsfunktion zum Einsatz.

remove_cont_app_handle – schließt eine offene Session der TCP-Anwendungsschnittstelle.

Argumente:	`c_seek_list_t *entry, char *msg`
Rückgabewert:	`int8_t`
Datei:	cont_ext.c, cont_ext.h

Vor dem Schließend des in `*entry` enthaltenen Socketdeskriptors wird die in `*msg` übergebene Nachricht versendet. Danach entfernt ein `detach_callback_func`-Aufruf die registrierte Behandlungsfunktion sowie den angegebenen Socketdeskriptor aus der Liste der zu überwachender Sockets in der Programmhauptschleife. Nach dem Beenden der Socketverbindung wird abschließend mittels eines `c_seek_list_remove`-Aufrufes der mit `*entry` übergebe Datensatz entfernt.

copy_rrep_ext – kopiert jedes einzelne Feld der empfangenen RREP-Kontextprotokollerweiterung in die vorgesehene Datenstruktur.

Argumente: `char *pkg, C_RREP_EXT *ext`
Rückgabewert: `int8_t`
Datei: cont_ext.c, cont_ext.h

Diese Funktion kopiert alle einzelnen Felder der empfangenen RREP-Kontextprotokollerweiterung (`*pkg`) in die bereitgestellte Datenstruktur (`*ext`). Das Kopieren der einzelnen Felder ist notwendig, da sich der Aufbau der verwendete Datenstruktur vom Typ `C_RREP_EXT` nicht vollständig mit der empfangenen Protokollerweiterung deckt. Die Funktion wird von der modifizierten Behandlungsroutine (`rrep_process`) für eintreffende RREP-Nachrichten aus der Datei `aodv_rrep.c` aufgerufen.

print_rrep_cont_nr – gibt die angegebenen Kontexttypen in RREP-Paketerweiterungen als lesbare Zeichenkette zurück.

Argumente: `C_RREP_EXT *ext`
Rückgabewert: `char *`
Datei: cont_ext.c, cont_ext.h

Die Funktion erzeugt eine lesbare Zeichenkette aller in `*ext` markierter Kontexttypen und gibt diese als Rückgabewert aus.

count_rrep_cont – gibt die Anzahl der angegebenen Kontexttypen in RREP-Paketerweiterungen zurück.

Argumente: `C_RREP_EXT *ext`
Rückgabewert: `u_int8_t`
Datei: cont_ext.c, cont_ext.h

Die Anzahl aller markierter Kontexttypen einer RREP-Kontextpaketerweiterung (`*ext`) wird als Rückgabewert ermittelt.

print_rreq_cont_nr – gibt die gesetzten Kontexttypen und Prioritäten einer RREQ-Paketerweiterungen als lesbare Zeichenkette zurück.

Argumente: `C_RREQ_EXT *ext`
Rückgabewert: `char *`
Datei: cont_ext.c

Die Funktion erzeugt eine lesbare Zeichenkette aller in `*ext` markierter Kontexttypen und entsprechenden Prioritäten und gibt diese als Rückgabewert aus.

c_seek_list_insert – speichert alle relevanten Informationen zu einer kontextsensitiven Dienstanfrage in einer verketteten Liste.

Argumente: `int sock, u_int16_t session, struct in_addr dest_addr, int ttl, C_RREQ_EXT *conreq, u_int8_t flags`
Rückgabewert: `c_seek_list_t *`
Datei: cont_seek_list.c, cont_ext.h

Die Funktion fügt die übergebenen Werte einer verketteten Liste von aktiven kontextbehafteten Dienstanfragen hinzu. Die Einträge dieser Liste werden beim Erhalt einer Antwort des Kontextrouters sowie nach Ablaufen des spezifizierten Time-out-Wertes konsultiert, um alle relevanten Informationen abzurufen.

c_seek_list_remove – entfernt den spezifizierten Eintrag zu einer kontextsensitiven Dienstanfrage aus der verketteten Liste.

Argumente: `c_seek_list_t *entry`
Rückgabewert: `int8_t`
Datei: cont_seek_list.c, cont_ext.h

Nach Abschluss einer Dienstanfrage oder dem Beenden der Schnittstellenverbindung zur Kontextanwendung werden zu zuvor abgelegten Informationen gelöscht und der belegte Speicher freigegeben.

c_seek_list_find – sucht nach einer aktiven kontextsensitiven Dienstanfrage.

Argumente: `const u_int16_t service, const u_int16_t session`
Rückgabewert: `c_seek_list_t *`
Datei: cont_seek_list.c, cont_ext.h

Mittels den übergebenen Parametern `service` und `session` kann nach einem Datensatz zu aktiven kontextbehafteten Dienstanfragen gesucht werden. Enthält das zweite Argument `session` den Wert „`0`“, bleibt es bei der Suche unberücksichtigt. Der Rückgabewert ist ein Zeiger auf den ersten gefundenen Datensatz, der mit den angegebenen Argumenten übereinstimmt.

c_seek_list_find_sock – sucht nach einem Datensatz zu aktiven kontextsensitiven Dienstanfragen anhand eines Socketdeskriptors.

Argumente: `int sock`
Rückgabewert: `c_seek_list_t *`
Datei: cont_seek_list.c, cont_ext.h

Die Funktion sucht anhand des übergebenen Socketdeskriptors nach einem Datensatz zu aktiven kontextsensitiven Dienstanfragen. Der Rückgabewert ist ein Zeiger auf den ersten gefundenen Datensatz, der mit dem angegebenen Argument übereinstimmt.

c_blacklist_insert – fügt eine IP-Adresse der Serverblackliste hinzu.
Argumente: `u_int32_t id`
Rückgabewert: ohne
Datei: cont_srv_blacklist.c, cont_srv_blacklist.h

Diese Funktion fügt die angegebene IP-Adresse (`id`) der Liste unerwünschter Dienstanbieter hinzu. Hierbei wird zum einem ein Datensatz in einer verketten Liste angelegt und zum anderen die Adresse einer Datei angehängt. Die Werte der verketteten Liste dienen der späteren Suche nach Einträgen. Die Datei wird zum permanenten speichern der Liste nach Programmende benötigt. Die in der Datei abgelegten IP-Adressen sind als gewöhnliche dezimalen Zahlen in *network byte order* gespeichert.

c_blacklist_remove – entfernt eine IP-Adresse aus der Serverblackliste.
Argumente: `c_server_blacklist_t *entry`
Rückgabewert: `int8_t`
Datei: cont_srv_blacklist.c, cont_srv_blacklist.h

Das Entfernen eines Eintrags aus der Liste unerwünschter Dienstanbieter erfolgt sowohl innerhalb der verketteten Liste als auch in der definierten Datei.

c_blacklist_find – sucht eine IP-Adresse in der Serverblackliste.
Argumente: `const u_int32_t id`
Rückgabewert: `c_server_blacklist_t *`
Datei: cont_srv_blacklist.c, cont_srv_blacklist.h

Die Suche nach einer übergebenen IP-Adresse (`id`) erfolgt ausschließlich in der angelegten verketteten Liste. Wird eine Übereinstimmung festgestellt, liefert der Rückgaberwert der Funktion einen Zeiger auf den entsprechenden Datensatz der Blackliste.

c_blacklist_init – initiiert die Serverblackliste.

Argumente:	ohne
Rückgabewert:	ohne
Datei:	cont_srv_blacklist.c, cont_srv_blacklist.h

Diese Funktion wird während der Programminitiierung von der Funktion `aodv_cont_ext_init` aufgerufen. Sie öffnet die Datei mit den Einträgen der Liste unerwünschter Dienstanbieter und erzeugt daraus eine zur Laufzeit verwaltete verkettete Liste von Einträgen der Serverblackliste.

c_blacklist_print – gibt die Einträge der Serverblackliste als lesbare Zeichenkette aus.

Argumente:	ohne
Rückgabewert:	`char *`
Datei:	cont_srv_blacklist.c, cont_srv_blacklist.h

Die Funktion erstellt eine für den Menschen lesbare Version der Liste unerwünschter Dienstanbieter. Hierzu wird ein Zeiger auf die erzeugte Zeichenkette als Rückgabewert geliefert.

c_route_discovery_timeout – Behandlungsfunktion von Time-out-Ereignissen der kontextsensitiven Dienstsuche.

Argumente:	`void *arg`
Rückgabewert:	ohne
Datei:	cont_timeout.c, cont_timeout.h

Nach Ablauf eines in AODV-UU registrierten Time-outs wird die jeweils spezifizierte Time-out-Behandlungsfunktion ausgeführt. Diese Funktion ist eine modifizierte Version der von AODV-UU bereit gestellten `route_discovery_timeout`-Funktion aus der Datei `aodv_timeout.c`. Sie sendet nach Ablauf der zuvor angegebenen Time-out-Zeitspanne das jeweilige CRREQ-Paket erneut. Die maximale Anzahl erneut gesendeter Dienstanfragen ist durch den Parameter *RREQ_RETRIES* begrenzt. Alle benötigten Informationen zum wiederholten Senden der Anfrage werden aus dem Datensatz vom Typ `c_seek_list_t` entnommen, welcher beim ersten Versenden der Anfrage erstellt wurde.

detach_callback_func – entfernt die Registrierung eines Socketdeskriptors und dessen Behandlungsfunktion für die Programmhauptschleife.

Argumente: `int fd, callback_func_t func`
Rückgabewert: `int`
Datei: main.c, defs.h

Diese Funktion wird immer dann aufgerufen, wenn eine zuvor registrierte Socketverbindung zur Kontextanwendung beendet wird. Somit entfällt die Überwachung des Sockets mittels des `select`-Aufrufes in der Hauptschleife.

A.2 Modifizierte Funktionen

Im folgenden Abschnitt sind die modifizierten AODV-UU-Funktionen genannt. Weiterhin wird eine kurz Erklärung der Veränderung angegeben. Zusätzlich sind alle durchgeführten Änderungen am Originalquellcode durch Kommentare kenntlich gemacht.

Funktion: **attach_callback_func**
Datei: main.c

Durch das dynamische Hinzufügen und Entfernen von Socketdeskriptoren in die Liste der von `select` überwachten Sockets musste diese Funktion verändert werden. Jetzt wird im `callbacks`-Array zuerst nach einem freien Element gesucht und anschließend die übergebenen Werte abgelegt.

Funktion: **usage**
Datei: main.c

Hier wurde der neue Kommandozeilenparameter „`-C`“ mit in die Hilfeausgabe des Programms aufgenommen.

Funktion: **main**
Datei: main.c

Neben der Änderungen der Initialisierung des `callbacks`-Array wurde der neue Kommandozeilenparameter „`-C`“ in die Liste gültiger Parameter aufgenommen. Die globale Variable `click_compatibly` repräsentiert den Staus des Parameters. Weiterhin werden die Funktionen `aodv_cont_ext_init` und `icmp_router_msg_init` nach den anderen Initialisierungsfunktionen ausgeführt.

Daneben wurde die Überprüfung der von `select` zurückgegebenen Deskriptorliste in der Hauptschleife geringfügig angepasst. Nun werden alle gültigen Einträge des `callbacks`-Array danach überprüft ob, sie in einen Socketdeskriptor enthalten, der

zum Lesen bereit ist.

Funktion: **rrep_process**
Datei: aodv_rrep.c

Diese Funktion führt eine Analyse der empfangenen RREP-Nachrichten durch. Daneben wird auch nach vorhandenen Protokollerweiterungen gesucht. So können durch ein `switch / case`-Konstrukt Aktionen für jeden spezifizierten Erweiterungstyp ausgeführt werden.

Nach Einfügen einer `case RREP_CONT_EXT:`-Klausel beginnt die Analyse der vom Kontextrouter empfangenen Protokollerweiterung zur kontextsensitiven Dienstsuche. Hierbei wird unter anderem überprüft, ob der angebotene Dienstanbieter in der Liste unerwünschter Server enthalten ist. Falls dieser Fall eintritt, wird eine erneute Dienstanfrage automatisch abgesetzt. Andernfalls erhält die anfragende Anwendung eine Nachricht über Erfolg oder Misserfolg der Dienstanfrage.

Funktion: **rreq_create**
Datei: aodv_rreq.c

Diese Funktion erstellt eine RREQ-Nachricht, hierbei wurde die Behandlung des neuen Flags „*No Reroute*“ hinzugefügt.

A.3 Schnittstellensyntax

Im Folgenden sind alle implementierten Schlüsselwörter der realisierten Anwendungsschnittstelle und deren Argumente sowie Gültigkeit aufgelistet.

context-request – leitet eine kontextbehaftete Dienstanfrage ein.
Argumente: ohne
Gültigkeit: nicht zusammen mit `blacklist-add`, `blacklist-del`, `blacklist-print`, `crreq-timeout`, `crreq-timeout-print`, `force-soli`, `help` und `quit`

Dieses Schlüsselwort ist vor den übermittelten Diensttyp und Kontextnummern anzugeben, da diese Befehle sonst ihre Gültigkeit verlieren.

no-reroute – setzt das *No Reroute*-Flag in allen zukünftigen CRREQ-Paketen.
Argumente: ohne
Gültigkeit: nur nach `context-request`

Der `no-reroute`-Befehl ist nur in Verbindung mit dem `context-request`-Schlüsselwort gültig. Das *No Reroute*-Flag signalisiert dem Kontextrouter, dass ein anfragender Client kein automatisches *Rerouting* der weitergeleitenen Anwendungs-IP-Pakete wünscht. Der Kontextrouter sucht im Normalfall einen alternativen Dienstanbieter, falls der momentan verwendete, nicht mehr erreichbar sein sollte. Dieses Verhalten wird als *Rerouting* bezeichnet. Somit ist es Aufgabe der Anwendung bei Ausfall eines Diensteanbieters, Alternativen zu suchen.

reroute – deaktiviert das *No Reroute*-Flag in allen zukünftigen CRREQ-Paketen.
Argumente: ohne
Gültigkeit: nur nach `context-request`

Dieser Befehl hat die komplementäre Wirkung zum `no-reroute`-Schlüsselwort. Somit findet ein automatisches *Rerouting* bei Bedarf durch den Kontextrouter statt.

blacklist-print – gibt alle IP-Adressen der Serverblackliste aus.
Argumente: ohne
Gültigkeit: nicht in Verbindung mit `context-request`

Der Routing-Daemon verwaltet eine Liste mit unerwünschten Dienstanbietern. Diese Liste wird mittels diesem Befehl ausgegeben. Erhält der Client eine Antwort infolge einer Dienstanfrage mit einem angebotenen Server auf der lokalen *blacklist*, wird selbständig eine neue Dienstanfrage gestellt. Hierbei trägt das *ID*-Feld des erneut versandten *CRREQ*-Paketes die unerwünschte IP-Adresse. Der Kontextrouter kann nun einen alternativen Dienstanbieter anbieten, sofern er vorhanden ist.

crreq-timeout-print – gibt die Zeitspanne aus, die verstreichen muss, bis ein CRREQ-Paket von einer Time-Out-Funktion erneut versendet wird.
Argumente: ohne
Gültigkeit: nicht in Verbindung mit `context-request`

Vor dem Versenden eines *CRREQ* wird stets eine Zeitspanne definiert, die verstreichen muss, bevor die Dienstanfrage erneut gesendet wird, falls kein Kontextrouter antwortet. Wurde kein Time-out-Wert durch den Nutzer festgelegt, gibt der Befehl den vom AODV-Routing-Daemon genutzten Standardwert aus.

force-soli – erzwingt das Senden einer *ICMP-Router-Solicitation*-Nachricht.
Argumente: ohne
Gültigkeit: nicht in Verbindung mit `context-request`

ICMP-Router-Solicitation-Nachrichten werden nach [Deb07] zum Auffinden von Kontextroutern genutzt. Diese werden im Normalfall selbstständig durch den Routing-Daemon versandt, falls keine Kontextrouteradressen bekannt sind. Dieser Befehl erzwingt das Versenden einer derartigen Nachricht.

help – gibt eine Liste aller verfügbaren Kommandos aus.
Argumente: ohne
Gültigkeit: nicht in Verbindung mit `context-request`

Ein versandtes `help`-Schlüsselwort liefert eine Liste mit allen implementierten Kommandos beziehungsweise Schlüsselwörtern.

quit – beendet die Verbindung zum TCP-Socket des Routing-Daemons.
Argumente: ohne
Gültigkeit: nicht in Verbindung mit `context-request`

Nach der Terminierung des TCP-Sockets beendet sich das Testprogramm `cont_client` ebenfalls.

service – spezifiziert den gewünschten Diensttyp.
Argumente: Nummer des gewünschten Dienstes
Gültigkeit: nur nach `context-request`

Mittels `service` wird nach dem `context-request`-Schlüsselwort die gewünschte Dienstnummer angegeben. Wie Kapitel 2.3 zu entnehmen ist, sind momentan durch das 16-Bit breite *Service*-Feld $2^{16}-1$ Dienstnummern möglich. Ohne Angabe des `service`-Schlüsselwortes kann keine Dienstanfrage abgesetzt werden.

ct – spezifiziert die gewünschten Kontexttypen sowie deren Prioritäten.
Argumente: gewünschte Kontextnummern und Prioritäten, nach Schema $ct, p : ct, p \ldots$
Gültigkeit: nur nach `context-request`

Die Argumente des `ct`-Schlüsselwortes spezifizieren die gewünschten Kontexttypen sowie deren Prioritäten. Hierbei können beliebig viele ct, p-Tupel mittels einem „:“-Zeichen aneinander gekettet werden. Wobei ct die Kontextnummer und p die dazugehörige Priorität angibt. [Deb07] definiert einen Wertebereich der Kontextnummern

von *Eins* bis *Einhundert* und Prioritäten zwischen *Null* und *Sieben*. Werden zu einer Dienstanfrage keine Prioritäten angegeben, sind alle entsprechenden Felder des *CRREQ*-Paketes auf *Null* gesetzt.

router – übergibt die IP-Adresse eines Kontextrouters.
Argumente: IP-Adresse des Kontextrouters
Gültigkeit: immer

Mittels dem `router`-Schlüsselwort kann ein vom Nutzer präferierter Kontextrouter angegeben werden. Eine so vom Routing-Daemon gespeicherte Kontextrouteradresse, wird nicht von eintreffenden *ICMP Router Advertisement*-Nachrichten überschrieben.

id – spezifiziert den *ID*-Wert des CRREQ-Paketes.
Argumente: IP-Adresse des Dienstanbieters
Gültigkeit: nur nach `context-request`

Mit Hilfe des `id`-Argumentes ist es möglich, den *ID*-Wert des nächsten *CRREQ*-Paketes anzugeben. Auf diese Weise kann das Verhalten der Blacklistfunktion des Routing-Daemons von Hand nachgeahmt werden. [Deb07] geht detailliert auf die Bedeutung des *ID*-Feldes der *CRREQ*-Nachrichten ein.

blacklist-add – fügt eine IP-Adresse der Serverblackliste hinzu.
Argumente: IP-Adresse des Dienstanbieters
Gültigkeit: nicht in Verbindung mit `context-request`

`blacklist-add` fügt die angegebene IP-Adresse der Liste der unerwünschten Dienstanbieters des Routing-Daemons hinzu.

blacklist-del – entfernt eine IP-Adresse aus der Serverblackliste.
Argumente: IP-Adresse des Dienstanbieters
Gültigkeit: nicht in Verbindung mit `context-request`

`blacklist-del` entfernt die angegebene IP-Adresse aus der Liste der unerwünschten Dienstanbieters des Routing-Daemons.

crreq-timeout – setzt die Time-Out-Zeitspanne der CRREQ-Pakete.
Argumente: ganzzahlige Zeitspanne in Millisekunden
Gültigkeit: nicht in Verbindung mit `context-request`

Nach der spezifizierten Zeitspanne wird das zuvor gesendete *CRREQ* erneut versandt. Der angegebene Time-Out-Wert beeinflusst lediglich *CRREQ*-Nachrichten. Auf gewöhnliche *RREQ*-Pakete hat dieser Parameter keinen Einfluss.

B Inhalt des Datenträgers

Auf dem beigefügten Datenträger befinden sich folgender Inhalt.

Verzeichnis	**Inhalt**
/dok_latex/	LaTeX-Dateien der Diplomarbeit
/dok_pdf/	Diplomarbeit im PDF-Format
/messungen/	Wireshark Capturefiles, AODV-UU-Logdateien sowie Programmausgaben aller durchgeführten Testszenarios
/software/aodv-uu-0.9.5/	AODV-UU, ohne Änderungen
/software/aodv-uu-context-0.9.5/	AODV-UU, mit Kontexterweiterungen
/software/aodv-uu-context-festnetz.0.9.5/	AODV-UU, mit Kontexterweiterungen, Festnetzversion
/software/cipping/	Modifiziertes Ping-Programm
/software/cont_client/	Testclient für Kontextanwendungsschnittstelle

Literaturverzeichnis

[Bra05] Braem, Bart: *Implementation and evaluation of ad-hoc on demand distance vector routing*, Universität Antwerpen, Masterarbeit, Juni 2005. `https://euterpe.cmi.ua.ac.be/~bbraem/thesis/thesis.pdf`

[CBR05] Chakeres, Ian ; Belding-Royer, Elizabeth: AODV Implementation Design and Performance Evaluation. In: *International Journal of Wireless and Mobile Computing (IJWMC)* (2005). `http://www.cs.ucsb.edu/~ebelding/txt/ijwmc05.pdf`

[Deb07] Debes, Maik: *Kontextsensitives Routing – Architekturkonzept –*. Technische Universität Ilmenau, Fachgebiet Kommunikationsnetze, Juni 2007. `http://www.db-thueringen.de/servlets/DocumentServlet?id=8439`

[Dee91] Deering, S.: *ICMP Router Discovery Messages*. RFC 1256 (Proposed Standard). `http://tools.ietf.org/html/rfc1256`. Version: September 1991 (Request for Comments)

[DL04] Debes, Maik ; Lewandowska, Agnieszka: *Definition von Kontextinformation*. Juni 2004 (TAS-Zwischenbericht)

[FSF07] Free Software Foundation, Inc.: *GNU GENERAL PUBLIC LICENSE*, Juni 2007. `http://www.gnu.org/licenses/gpl-3.0.txt`

[HJ01] Hu, Yih-Chun ; Johnson, David B.: Implicit Source Routing in On-Demand Ad Hoc Network Routing. In: *Proceedings of the Second Symposium on Mobile Ad Hoc Networking and Computing* (MobiHoc 2001), 2001, 1-10

[JHM07] Johnson, D. ; Hu, Y. ; Maltz, D.: *The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4*. RFC 4728 (Experimental). `http://tools.ietf.org/html/rfc4728`. Version: Februar 2007 (Request for Comments)

[JM96] Johnson, David B. ; Maltz, David A.: Dynamic Source Routing in Ad Hoc Wireless Networks. Version: 1996. `http://citeseer.ist.psu.edu/johnson96dynamic.html`. In: Imielinski (Hrsg.) ; Korth (Hrsg.): *Mobile Computing* Bd. 353. Kluwer Academic Publishers, 1996

[KMC+00] Kohler, Eddie ; Morris, Robert ; Chen, Benjie ; Jannotti, John ; Kaashoek, M. F.: The click modular router. In: *ACM Trans. Comput. Syst.* 18 (2000), Nr. 3, 263–297. `http://doi.acm.org/10.1145/354871.354874`. – ISSN 0734–2071

[KU03] Kuladinithi, K. ; Udugama, A.: *JAdhoc System Design Manual*. University of Bremen, ComNets, Deutschland, 2003. `http://www.comnets.uni-bremen.de/~koo/JAdhoc.pdf`

[KZG03] Kawadia, Vikas ; Zhang, Yongguang ; Gupta, Binita: System Services for Implementing Ad-Hoc Routing: Architecture, Implementation and Experiences. In: *Proceedings of MobiSys 2003: The First International Conference on Mobile Systems, Applications, and Services*, 2003, 99-112

[LDS05] Lewandowska, Agnieszka ; Debes, Maik ; Seitz, Jochen: An Architecture for Context-Sensitive Telecommunication Applications. In: *WEBIST 2005, Proceedings of the First International Conference on Web Information Systems and Technologies, Miami, USA*, 2005, S. 40–47

[LNT02] Lundgren, H. ; Nordström, E. ; Tschudin, C.: Coping with communication gray zones in IEEE 802.11b based ad hoc networks. In: *WOWMOM '02: Proceedings of the 5th ACM international workshop on Wireless mobile multimedia*. New York, NY, USA : ACM Press, 2002. – ISBN 1–58113–474–6, 49–55

[Pas05] Paschold, Guido: *Entwurf und Simulation eines kontextsensitiven Routingprotokolls*, Technische Universität Ilmenau, Fakultät für Elektrotechnik und Informationstechnik, Diplomarbeit, 2005

[PB94] Perkins, Charles ; Bhagwat, Pravin: Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In: *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, 1994, 234–244

[PBRD03] Perkins, C. ; Belding-Royer, E. ; Das, S.: *Ad hoc On-Demand Distance Vector (AODV) Routing*. RFC 3561 (Experimental). `http://tools.ietf.org/html/rfc3561`. Version: Juli 2003 (Request for Comments)

[PD04] *Kapitel* 4.2. In: Peterson, Larry L. ; Davie, Bruce S.: *Computernetze: eine systemorientierte Einführung*. Dpunkt Verlag, Heidelberg, 2004

[Ren06] Renhak, Karsten: *Router mit Kontextwissen*, Technische Universität Ilmenau, Fakultät für Elektrotechnik und Informationstechnik, Studienarbeit, August 2006

[RP00] Royer, E. ; Perkins, C.: Multicast Ad hoc On-Demand Distance Vector (MAODV) Routing / Internet Engineering Task Force. Version: Juli 2000. `http://tools.ietf.org/html/draft-ietf-manet-maodv-00`. 2000. – Internet Draft. – Work in progress

[Wen07a] Wenzel, Marco: *Evaluierung eines modularen Routers und Implementierung eines Ad-hoc-Routingprotokolls*, Technische Universität Ilmenau, Fakultät für Elektrotechnik und Informationstechnik, Studienarbeit, Januar 2007

[Wen07b] Wenzel, Marco: *Konzeption und Umsetzung eines Demonstrators zur Verifikation kontextsensitiver Routingprotokolle*, Technische Universität Ilmenau, Fakultät für Elektrotechnik und Informationstechnik, Diplomarbeit, September 2007

[Wik07] Wikipedia: *List of ad-hoc routing protocols — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=List_of_ad-hoc_routing_protocols&oldid=138077853`. Version: 2007. – [Online; accessed 9-July-2007]

[Wol06] Wolf, Jürgen: *Linux-Unix-Programmierung*. Galileo Computing, Bonn, 2006. – ISBN 9783898427494

Softwareverzeichnis

[1] *The Click Modular Router Project.* http://www.read.cs.ucla.edu/click

[2] *Gentoo-Linux.* http://www.gentoo.org

[3] *Git – Fast Version Control System.* http://git.or.cz/

[4] *GNU C Library.* http://www.gnu.org/software/libc/

[5] *GNU Compiler Collection.* http://gcc.gnu.org/

[6] *IP OPTION NUMBERS.* http://www.iana.org/assignments/ip-parameters

[7] *Multicast Extensions of AODV(MAODV).* http://www.hynet.umd.edu/research/maodv/MAODV-UMD.html

[8] *The netfilter.org project.* http://www.netfilter.org/

[9] *The Network Simulator – ns-2.* http://nsnam.isi.edu/nsnam/index.php/Main_Page

[10] *OpenSSH.* http://www.openssh.com/

[11] *OpenVPN.* http://openvpn.net/

[12] *Tcpdump.* http://www.tcpdump.org

[13] *VTun.* http://vtun.sourceforge.net/

[14] *Wireshark.* http://www.wireshark.org/

[15] Chakeres, Ian: *UCSB AODV Implementation.* http://moment.cs.ucsb.edu/AODV/aodv.html#Implementations

[16] Kawadia, Vikas ; Zhang, Yongguang ; Gupta, Binita: *AODV–UIUC.* http://aslib.sourceforge.net/

[17] Klein-Berndt, Luke: *Kernel-AODV*. http://w3.antd.nist.gov/wctg/aodv_kernel/

[18] Nordström, Erik: *AODV-UU*. http://core.it.uu.se/core/index.php/AODV-UU

[19] Tourrilhes, Jean: *Wireless Tools for Linux*. http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html

[20] Udugama, Asanga: *UoBWinAODV*. http://www.aodv.org/

[21] Udugama, Asanga ; Pangboonyanon, Varaporn ; Kuladinithi, Koojana ; Daniels, Brice: *UoB-JAdhoc*. http://www.aodv.org/

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis und Formelzeichen

AODV ..... **A**d hoc **O**n-Demand **D**istance **V**ector Routing
API ..... **A**pplication **p**rogramming **i**nterface
ARP ..... **A**ddress **R**esolution **P**rotocol
ASCII ..... **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
ASL ..... **A**d-hoc **S**upport **L**ibrary
BGP ..... **B**order **G**ateway **P**rotocol
CPU ..... **C**entral **P**rocessing **U**nit
CRREP ..... **C**ontext **R**oute **Rep**ly Message
CRREQ ..... **C**ontext **R**oute **Req**uest Message
CVS ..... **C**oncurrent **V**ersions **S**ystem
DSDV ..... **D**estination-**S**equenced **D**istance-**V**ector Routing
DSR ..... **D**ynamic **S**ource **R**outing
GNU ..... **G**NU is **n**ot **U**nix
GUI ..... **G**raphical **U**ser **I**nterface
I/O ..... **I**nput **O**utput (Ein-/Ausgabe)
IANA ..... **I**nternet **A**ssigned **N**umbers **A**uthority
ID ..... **Id**entifikationsnummer
IEEE ..... **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
IOCTL ..... **I**nput/**O**utput **C**on**t**ro**l**
IP ..... **I**nternet **P**rotocol
IPv6 ..... **I**nternet **P**rotocol **V**ersion **6**
ITEF ..... **I**nternet **E**ngineering **T**ask **F**orce
JRE ..... **J**ava **R**untime **E**nvironment
LAN ..... **L**ocal **A**rea **N**etwork
MAC ..... **M**edia **A**ccess **C**ontrol
MAODV ..... **M**ulticast **A**d hoc **O**n-Demand **D**istance **V**ector Routing
NDIS ..... **N**etwork **D**evice **I**nterface **S**pecification

OSPF	**O**pen **S**hortest **P**ath **F**irst
PC	**P**ersonal **C**omputer
PCMCIA	**P**ersonal **C**omputer **M**emory **C**ard **I**nternational **A**ssociation
PDA	**P**ersonal **D**igital **A**ssistant
RERR	**R**oute **Err**or Message
RIP	**R**outing **I**nformation **P**rotocol
RREP	**R**oute **Rep**ly Message
RREP-ACK	**R**oute **Rep**ly **Ack**nowledgement
RREQ	**R**oute **Req**uest Message
SVN	**S**ub**v**ersio**n**
TAS	**T**ouristisches **A**ssistenz**s**ystem für barrierefreie Urlaubs-, Freizeit und Bildungsaktivitäten
TCP	**T**ransmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol
VM	**V**irtual **M**achine
VPN	**V**irtual **p**rivate **n**etwork
WLAN	**W**ireless **L**ocal **A**rea **N**etwork

Thesen zur Diplomarbeit

1. Der personalisierte Informationsaustausch erlangt in der heutigen Gesellschaft zunehmend an Bedeutung.
2. Das kontextsensitive Routing dient der individuell angepassten Dienstnutzung in paketvermittelten Netzen.
3. Eine Implementierung des reaktiven Routingprotokolls AODV verlangt die Behandlung verschiedener Ereignisse, die nicht von Routingarchitekturen gewöhnlicher Betriebssysteme erkannt werden.
4. Die Erweiterung des AODV Routingprotokolls übermittelt zusätzliche Daten, die bei der Entscheidung über die Wegewahl von kontextsensitiven Routingmechanismen herangezogen werden.
5. Die Implementierung *AODV-UU* der schwedischen Universität in Uppsala stellt die Grundlage für realisierte Erweiterungen am Routingprotokoll AODV dar.
6. Die entworfene und implementierte Schnittstelle kann zum manuellen Testen sowie von möglichen Anwendungen genutzt werden, um kontextsensitive Dienste in Anspruch zu nehmen.
7. Mit dem vorhandenen Demonstrator können die entwickelten Routingprotokolle verifiziert werden.
8. Messungen oder Funktionstests über ein *shared Medium*, wie es WLAN darstellt, sind stark von der momentanen Auslastung des Übertragungsmediums abhängig.
9. Mögliche Fehler in den Softwarekomponenten der Demonstratorumgebung lassen sich durch systematisches Testen ausfindig machen.
10. Der frei verfügbare Protokollanalysator *Wireshark* kann als Nachweis über die erfolgten Kommunikationsprozesse der beteiligten Komponenten dienen.

Ilmenau, den 03. 03. 2008 Karsten Renhak

Erklärung

Die vorliegende Arbeit habe ich selbstständig ohne Benutzung anderer als der angegebenen Quellen angefertigt. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer oder anderer Prüfungen noch nicht vorgelegt worden.

Ilmenau, den 03. 03. 2008 Karsten Renhak